Execution of automatically parallelized APL programs on RP3

by W.-M. Ching
D. Ju

We have implemented an experimental APL/C compiler, which accepts ordinary APL programs and produces C programs. We have also implemented a run-time environment that supports the parallel execution of these C programs on the RP3 computer, a shared-memory, 64-way MIMD machine built at the IBM Thomas J. Watson Research Center. The APL/C compiler uses the front end of the APL/370 compiler and imposes the same restrictions, but requires no parallelization directives from the user. The run-time environment is based on simple synchronization primitives and is implemented using Mach threads. We report the speedups of several compiled programs running on RP3 under the Mach operating system. The current implementation exploits only data parallelism. We discuss the relationship between the style of an APL program and its expected benefit from the automatic parallel execution provided by our compiler.

Introduction

During the past decade, there has been a tremendous amount of research on parallel processing intended to speed up the execution of applications. Confronted with a new parallel machine, a programmer usually has to prepare the application program by hand for parallel execution, using either a new parallel-programming language or some conventional language with added parallel constructs. The reported successful utilization of the Connection Machine [1] and of the GF-11 [2] has been achieved in this way. It is interesting to note that these two parallel machines are both SIMD (single-instruction-stream, multiple-data-stream) type.

The prospect of the extra work in partitioning an application for parallel execution and orchestrating the communication among the segments is daunting enough to dampen the ordinary user's enthusiasm for parallelism, no matter how easy such a process is considered by researchers in parallel programming. For the ordinary user, parallelism is interesting *only* if it is transparent.

A more ambitious approach is to develop a compiler for a high-level programming language, with the ability to

decompose and schedule a user's program for parallel execution. The language can be either a new programming language designed explicitly for parallel programming, such as Crystal [3], Id [4], and SISAL [5], or a conventional uniprocessor programming language. Notable research projects taking the latter approach are those of PTRAN [6], developed at the Thomas J. Watson Research Center, and Parafrase, in the Cedar Project [7] at the University of Illinois. Both currently focus on FORTRAN and on MIMD (multiple-instruction-stream, multiple-data-stream) machines with shared-memory architectures. PTRAN analyzes programs interprocedurally [8] to find all potential parallelism and later uses a process-former [9] to form parallel processes. There are also academic efforts that rely on new parallel-programming languages with a functional language flavor [3]. We observe that one weakness of such an approach is that new languages generally have few application programs available for testing.

This paper presents the result of our work on an experimental APL compiler that transforms programs written in conventional APL to programs that may be executed in parallel. No directives for parallel execution are required. Our compiled code runs on the RP3 system, the Research Parallel Processing Prototype, a multipurpose, shared-memory, 64-way MIMD machine built at the IBM Thomas J. Watson Research Center for doing research on parallel processing [10]. APL is widely used in many application areas, including financial analysis, engineering, and scientific computations. Because of the high level of the APL language and its array orientation, it is quite easy to determine which parts may be executed in parallel [11]. The succinctness of a well-written APL program implies that there are no incidental or artificial constraints on the order of execution. These constraints may well occur in FORTRAN programs.

We have chosen to implement a parallelizing APL compiler, but we feel our research is applicable to any high-level language that allows manipulation of large data structures, including database languages. While conceptually simple, our approach requires the implementation of many system-specific details. The effectiveness of this approach is actually due to the inducement the APL language provides to its users to write high-level programs focused on arrays and array manipulation.

Traditionally, APL processors have been implemented as interpreters, which carry the overhead of source-code interpretation during execution. Hence, we believe that work to produce good compilers that eliminate the interpretative overhead (which is sequential in nature) should precede work on parallel APL execution. Our previous work on an APL compiler [12] has prepared us well for this undertaking. While it is expensive to construct an APL compiler because of the very high level of the language, for the same reason the additional cost of parallelization (not counting the detailed work required to interface to a parallel machine) is relatively cheap once a sequential compiler exists.

In order to perform parallel-APL experiments on RP3, we first modified our APL/370 compiler [12], which generates System/370™ assembly code directly, to generate C source code. This C code is then compiled into ROMP machine code (the RP3 processor, which is also used in the IBM RT PC® [13]) by the Metaware Corporation C compiler for the RT.® Note that this is not a C compiler for the RP3 system, since it only generates code for the individual processors. Our compiler, called APL/C, accepts VS APL programs without requiring declaration or parallelization directives. The run-time environment for executing compiled APL/C code in parallel on the RP3 system is similar to but simpler than that used for the PTRAN system [14] on RP3. Mach-kernel threads [15] are used to implement parallel execution streams. Fetch-and-(op) instructions [13] are used to implement the simple synchronization primitives (fork and join) needed to handle the data parallelism (i.e., parallelism from operations on array elements) that exists in a typical APL program. We chose to use Mach-kernel threads instead of C-thread [15] to reduce the run-time overhead. We note that there has been work on translating APL to C [16–18] and APL to FORTRAN [19]. Our work here concentrates on issues unique to generating code for highly parallel machines and their run-time environments.

Our APL/C compiler is currently incomplete: Not all primitive functions or all their functionalities are implemented; our work is confined to exploiting data parallelism at the primitive function level; and all implemented primitive functions are not parallelized. Nevertheless, an existing group of APL programs, including ones to find primes and solve a Poisson equation and a financial application, have been compiled and executed on RP3. We present measurements of their parallel execution time and speedups. We note that this work of exploiting data parallelism in APL programs can also be carried out on SIMD machines such as the GF-11 computer [2]. However, we expect the cost of implementing a parallelizing compiler for the GF-11 to be very expensive because of the wide instruction words and deep pipelines used in the machine.

In the following section, we describe the APL/C compiler and the minor language restrictions it imposes. Next, we briefly describe the relevant features of the RP3 machine and its Mach operating system. We then describe our implementation of a run-time environment for parallel execution. This is followed by a description of measurements of parallel execution of example programs on the RP3. Finally, we discuss how parallel

speedup* depends on program style, and we compare this with the way compiler improvement over interpreters depends on program style. We also discuss how some machine features and our method of exploiting data parallelism limit the speedups.

The APL/C compiler

Our previous efforts [11] aimed at demonstrating parallel APL execution on a multiple-processor IBM 3090™ computer under the MVS operating system. To demonstrate the execution of automatically parallelized APL on the highly parallel experimental RP3 machine, we first had to adapt our compiler to generate code for the RP3 processors. We decided to use C because a C compiler was available for the RP3 processors as well as many other microprocessors.

The APL/C compiler uses the same front end as the APL/370 compiler; hence, it imposes the language restrictions stated in [12]. (The reasons why these restrictions are imposed are given in that paper.) The restrictions are as follows:

- The execute function, ⍎, is excluded.
- No system function is supported.
- Of the APL system variables, only $\square AV$ and a limited choice of $\square IO$ [12] are supported.
- Branch expressions are limited to the most common forms.
- A variable cannot change its storage type (which must be either numeric or character) or rank (which is limited to ≤ 7).

In short, the compiler covers most VS APL language features without requiring declaration, and maintains strict semantics of APL with regard to its order of execution. This is in sharp contrast to the compiler of Budd [16] and that of Bernecky et al. [17] (in the latter, no character data are allowed and all variables are treated as real numbers). Our compiler does not support any of the newer features incorporated in APL2™ [20] that are not in VS APL. Some of these new features, such as complex numbers, the "each" operator, and grouped parameters, are very desirable for parallel applications and can easily be accommodated with more work on the compiler. On the other hand, the nested-array feature of APL2 will require far more extensive work.

The compiler front end performs syntax analysis, parsing, and an interval-based dataflow analysis [12]. The result of dataflow analysis, *use-definition chaining*, is utilized in a "type-shape" analysis of the program variables used. This analysis determines the types and dimensions, known or unknown at compile time, of all variables used, except the parameter(s) to the main function in the compilation unit. Thus, no variable declaration is needed other than the specification of certain attributes of the outermost parameter(s) during the compiler invocation (see Section 2 of [12]). The compiler performs a live analysis of variables (indicating when a variable is redefined or no longer needed) and inserts temporary variables for storing intermediate values. We refer the reader to [12] for issues concerning the front end of the compiler that are not directly related to the parallel execution of APL.

The compiler front end also does a data-dependency analysis within each basic block [11]. This knowledge of data dependency is used by the front end to insert send/wait synchronization flags at parse-tree nodes to help the back end exploit functional parallelism (the certified potential to simultaneously execute independent subtrees in a basic block). Hence, the APL programs are automatically parallelized at the source-code basic-block level by the front end. However, the work reported here concentrates mainly on the data parallelism of array operations in APL primitives. This is because of our limited human resources for implementation—not because functional parallelism is not worth exploiting. In fact, it appears that the parallel speedup of some APL programs is limited when this functional parallelism is not exploited. This point is discussed below.

The back end of the APL/370 compiler was rewritten to generate C source code, while the basic structure of the back end remains the same. This includes the representation of source-program variables as well as the structure of several main functions of the back end. In particular, the function TREELIST, which evaluates a parse tree and calls various subfunctions to generate code for primitives encountered in traversing the parse tree, is almost unchanged. A program variable is determined by the front end of the compiler to be one of the following types: Boolean, integer, floating-point, or character. It is either a scalar, an array with dimensions known at compile time, a vector of unknown length, or an array with unknown dimensions. The subfunctions in the back end that implement primitive functions typically

- Check compatibility of shapes (dimensions) of their operands.
- If required, allocate space in the heap for the target variables.
- Carry out element-wise operations to produce target values.

The first two are clearly sequential operations, and the third is what usually constitutes data parallelism, or what corresponds to a DO-ALL loop in FORTRAN. We note that, in contrast to the situation with the interpreter, no

*Speedup is execution time on one processor divided by execution time when the program is run on multiple processors.

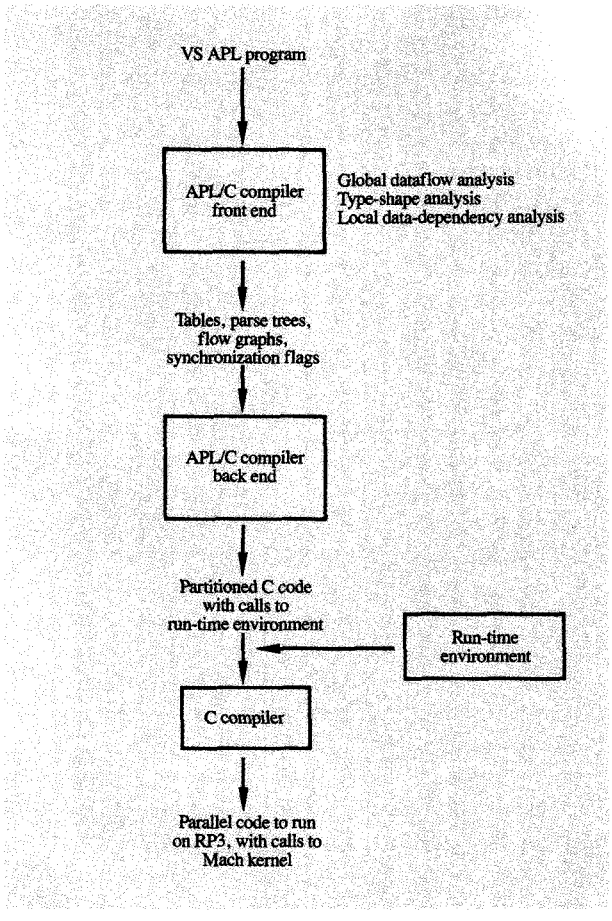


Figure 1

The parallelizing APL compiler for RP3.

type-satisfiability check is ever done at run time, and the shape-compatibility check is performed at run time only if it cannot be determined at compile time.

As stated above, at the time of this research on RP3, the implementation of our experimental APL/C compiler was not complete. Several APL primitive functions, including ⍋, ⍒, ⊖, ⌹, ⊤, and ⊥, were not implemented, and a number of cases within some implemented primitives and derived functions, such as reductions on Boolean variables and catenation of arrays of more than two dimensions, were not supported. Also, not all primitive functions have been given a parallel implementation. In particular, indexed assignment and general indexing were not parallelized, even though we believe it would be very desirable to do so. The parallelized primitives include most scalar functions, membership, inner product, and outer product. This highlights the peculiar character of APL: The power of APL and the ease of APL-program parallelization depend greatly on the availability of a large number of high-level primitives, but this is precisely the reason that a full compiler implementation of the language is very expensive. However, we do not require a full implementation to show interesting experimental results.

We also remark that the final efficiency of our generated code depends greatly on the quality of the C compiler we use. Fortunately, for our purpose of experimenting with parallelism, the absolute execution time is not our primary concern. We are most interested in the parallel speedup, which is basically not affected by the performance of the C compiler. We acknowledge that compilation time is important, since it is the cost of generating the code. From [21] we know that the APL/370 compilation time is dominated by the back end. We observed that the APL/C back end, on average, requires only about a fifth of the time of the APL/370 back end. This, of course, does not include the compilation time of the C compiler. The APL/C compiler is written in VS APL and is driven by an APL interpreter (i.e., it has not yet compiled itself), while the run-time environment is written in C. We illustrate the input/output flow of the components of the compiler system in **Figure 1**.

RP3, Mach, and the run-time environment of the APL/C compiler

Compiler research on automatic parallelism has generally been on MIMD, shared-memory machines rather than distributed-memory machines, because the former provide a single-address-space programming paradigm. The RP3 system, unlike many other MIMD machines, has shared memory and a large number of processors. This makes it suitable for experiments concerning scalability of parallel schemes. Each RP3 processor-memory element (PME) contains a processor, eight megabytes of memory, and a 32-kilobyte cache [21]. The PMEs are connected by an Omega network, and the memories are globally addressable and shared by all processors. Access by a processor in one PME to memories in other PMEs goes through the network and incurs a certain amount of performance penalty. (We refer the reader to [10, 22] for a general discussion of RP3 and its architecture.) In general, we find the RP3 network to be very fast in comparison with its processor speed, and usually not a bottleneck. The RP3 cache uses the write-through policy; i.e., modified entries are written back to memory. RP3 hardware does not maintain cache coherence. Hence, in order that programs be executed correctly, software on RP3, either produced by the programmer or the compiler, is responsible for ensuring that storage references are consistent. Software must explicitly designate storage as cacheable or noncacheable on a page-by-page basis and is responsible for flushing processor caches when necessary. The C compiler that we use makes everything cacheable by default. This works well, except for a peculiar problem

```
for (u0=0; u0<i; u0++)          /* i, n, and v3: matrix dimensions */
{q0=u0*n;                       /* u0, u1, and u2: loop indices    */
 q1=u0*v3;                      /* q0 and q1: matrix index vars.   */
for (u1=0; u1<n; u1++)          /* p2, lo2, and ro2: pointers to   */
{d=0;                           /*   target, left op, and right op */
for (u2=0; u2<v3; u2++)         /* d: accumulator                  */
d += lo2[q1+u2]*ro2[u1+u2*n];
p2[q0+u1] = d;}}
```

Figure 2

Core of APL inner product translated into C code.

related to the RT floating-point unit, which we discuss below. The storage-mapping software also must specify, on a page-by-page basis, whether storage is to be accessed sequentially (which is useful for local data) or interleaved.

The RP3 machine runs a modified version of the Mach operating system [23], which is a UNIX®-based system originally developed at Carnegie Mellon University [15]. Tasks in the original UNIX have separate address spaces. In contrast, a thread, a basic unit of concurrent execution in Mach, can share an address space with other threads in a task. Hence, multiple threads sharing a single address space can execute a particular computation in parallel. This matches the RP3 architecture and our parallel APL execution model. The Mach operating system provides a run-time library, called the C-thread package, which facilitates concurrent programming in C. It is flexible and convenient to use. However, we chose to use Mach-kernel threads to implement parallel program execution for reasons of efficiency. These threads are managed by our run-time environment because of performance considerations, even though this approach requires more implementation effort. The Mach operating system has been modified for the RP3 system to enable lightweight (low-overhead) kernel threads to be bound to processors and to provide memory and cache-management mechanisms.

This work describes the exploitation of data parallelism in basic blocks of APL source programs. Hence, the parallel execution model for our compiled APL code becomes a very simple *fork-join* model. We assume that all of our compiled APL code is sequentially executed except for *pure* array operations. As described in the previous section, the third part in the generated code for a primitive (or derived) function operating on array operands (vectors are one-dimensional arrays) is a pure array operation. All operations on scalars or one-element vectors, as well as space allocation and dimension-length checking of arrays, are considered to be sequential code segments and are executed serially. Since there is overhead in starting parallel execution, even a pure array section may not be suitable for parallel execution because of the small amount of data. During run time, we insert a size check at the beginning of an array operation to eliminate this problem. Only those arrays determined to be large enough (i.e., their execution time is larger than the overhead that would be incurred in splitting them into parallel tasks) are executed in parallel. If the size of an array operation is determined at run time to be over the threshold of parallel profitability, such a check can be eliminated. Since APL allows dynamic arrays, such checks at run time are the norm rather than the exception. The cost of a run-time check is, in general, insignificant.

We define a *parallel block* to be a code segment (of a pure array operation) judged to be large enough for parallel execution. For simplicity, we assume that a parallel block can always be divided into data-independent pieces of equal size (except for the last piece). We call these pieces "slices." Consequently, some derived functions such as reduction (the reduction $+/V$ sums all elements in vector V) are not parallelized, while others, such as inner product, are parallelized. We note that an inner product consists of a sequence of reductions. What we do here is parallelize the outer loops while giving each processor a number of iterations of the inner loop. The code for inner product is given in **Figure 2**. The compiler parallelizes only

the outermost *for*-loop, while the innermost *for*-loop, which represents the reduction, is not partitioned among different processors. This is a common practice in parallel programming for machines with numbers of processors less than one hundred. For massively parallel machines, it is very desirable to parallelize the reduction operation in a logarithmic number of steps [24]. The monadic function ⍳ is also excluded from parallelization.

At the entry of a parallel block, the execution is forked into parallel threads, each executing one slice of the parallel block. Upon completion of the parallel block, all threads are joined, and sequential execution resumes. To implement this simple run-time model, a main thread and multiple helper threads are created during initialization, and each thread is bound to a processor. The run-time environment performs this initialization by using system calls. The main thread executes the sequential portion of the code on one processor while all helper threads remain spin-waiting. When execution of the main thread reaches a parallel block, it assigns slices of the parallel block code to the helper threads and puts them to work in parallel on the computational job of the parallel block. The main thread also executes a slice of the parallel block. When a helper thread reaches the end of a parallel block, it notifies the main thread and goes back to spin-waiting. When the main thread completes its parallel slice, it spin-waits until it receives acknowledgment of completion from all its helper threads and then resumes the execution of the sequential section of the code.

We partition a parallel block in the compiled APL/C code into parallel slices with a scheduling policy usually called *chunk-scheduling*. The size of each slice, or chunk, is the total number of iterations divided by the number of available processors. (In the simple case of a scalar arithmetic function, the number of iterations is the number of elements in the array being manipulated, and in the case of an inner product such as matrix multiplication, it is the row size of the left operand.) We form parallel processes, i.e., assign threads dynamically to a parallel task in Mach on RP3. We do not depend on profiling information (estimated characteristics of the program behavior) to estimate the possible size of a parallel block. (There is a serious difficulty with using profiling data to decide the profitability of creating parallel processes for a portion of a program: Since profiling is usually done on sequential machines with limited runs, a projection of time to large problems may not be useful on programs with large data and of a more dynamic nature. This is similar to the situation in which observation of the inputs and outputs of a complicated mathematical function for a few points may not be sufficient to determine whether the output grows with the square or cube of the input.) Experimental results for other scheduling policies for our run-time environment on the RP3 can be found in [25].

The main thread activates its helper threads serially and then assigns itself the last slice of the parallel block. Distributed locks (in different RP3 PMEs) are used for synchronization in order to avoid memory contention. Each idle helper thread periodically checks its own lock to see whether any work is waiting. The join operation is implemented by a single shared variable. Each helper thread increments the shared variable to acknowledge completion of its work. The main thread checks this shared variable to ensure that all helper threads have finished. The assembly-language routine that implements synchronization uses the *fetch-and-add* instruction unique to the RP3. (For a comparison of distributed-lock and central-lock schemes and experimental data supporting the adoption of the distributed-lock scheme for our run-time environment, see [25]. We note that the distributed-lock scheme reduces memory contention and network traffic.) To activate the helper threads, the main thread calculates the lower and upper bounds of iteration indices and passes this information, together with the starting address of a subroutine embodying the work of a parallel chunk, to helper threads, and unlocks their locks one at a time. The use of a subroutine call format is purely for convenience in passing the necessary information to threads, and is relatively efficient in C.

All helper threads and the main thread operate on disjoint data (except for the locks and the shared variable for synchronization), so in principle all data can be declared cacheable without danger of inconsistency. However, floating-point support of the C compiler we used is implemented in a peculiar way: When a floating-point instruction is encountered for the first time, the compiler generates a branch to a location in the user program data area, where instructions to carry out the floating-point operation have been written. This requires that we declare all data areas (but not instruction areas) in our program to be noncacheable if a floating-point operation is present in the program. This obviously degrades the parallel performance of the code. We refer to [25] for relevant performance data related to this issue.

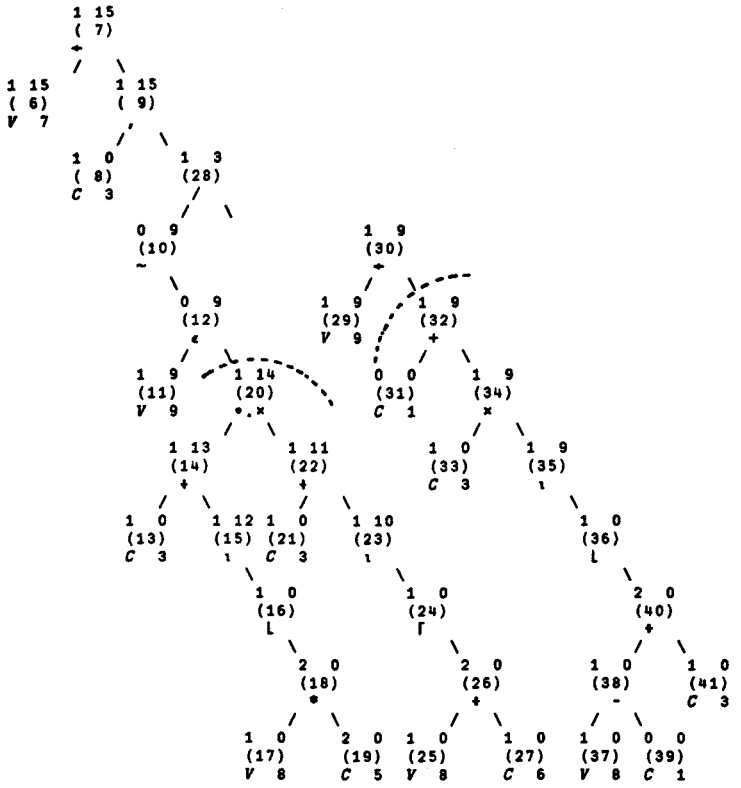
We do not support nested parallelism (i.e., splitting one parallel process further into a group of parallel subprocesses in our run-time environment, in contrast to PTRAN [14]. This is because we concentrate on data parallelism in APL, which does not require nested parallelism.

Speedups of automatically parallelized APL programs on the RP3

In this section we present the speedup and execution-time data for three APL programs. The three programs are not large, but neither are they trivial. They are interesting enough to illustrate why our current implementation of a simple run-time execution environment turns out to be reasonably effective for APL-style programs (we discuss

```
Z←2,(~V∈(2↓⍳⌊N*0.5)∘.×2↓⍳⌈N+3)/V←1+2×⍳⌊(N-1)÷2
```

(a)



(b)

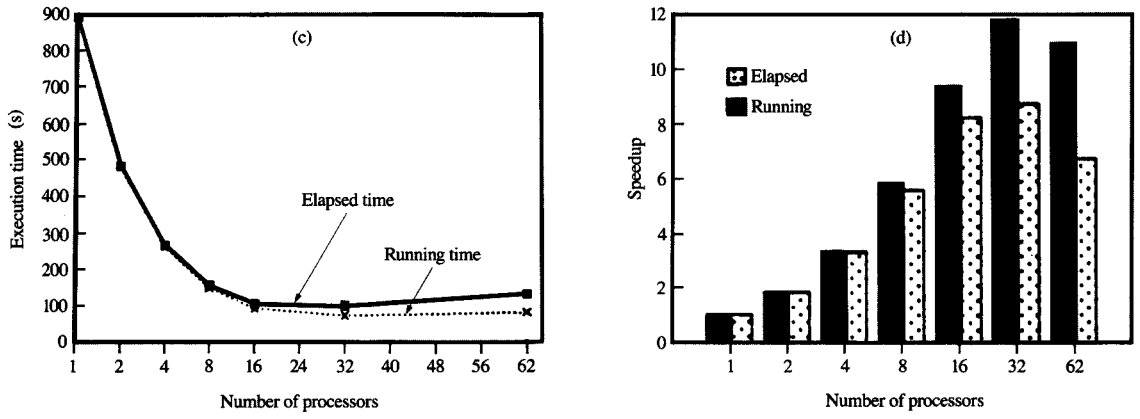


Figure 3

PRIME program: (a) the program itself; (b) parse tree (dashed lines delineate subtrees that can be executed in parallel); (c) execution time for $N = 2000$; (d) speedup for $N = 2000$.

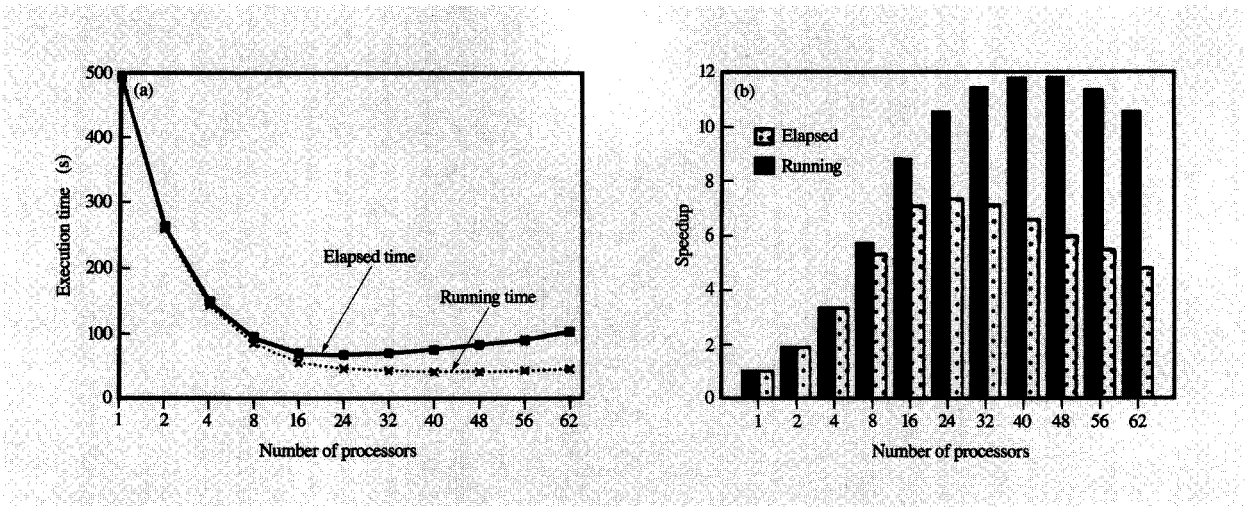


Figure 4

POISSON program for a 62 × 62 grid: (a) execution time; (b) speedup.

this point later in this section). In the results presented in Figures 3–5, elapsed time includes the time spent on initialization, whereas running time does not. The speedup is the ratio of running (elapsed) time on a single processor to running (elapsed) time on multiple processors for parallel execution. The RP3 machine was built for parallel software research on a real machine, not as a performance machine. By today's standards, ROMP/RT is a fairly slow processor, and the floating-point chip is not ideally integrated into the processor. Each program is described below; then the results of the experiments are discussed.

The first program, PRIME, with input N, computes all prime numbers up to N by means of a sieve method. It sets up a multiplication table large enough to check the membership of all odd numbers up to N. The number 2 and all odd numbers not in the table are primes. The source code is shown in **Figure 3(a)**, and its parse tree in **Figure 3(b)**. Experimental results are displayed in **Figures 3(c)** and **3(d)**. [Note that the horizontal scales in Figures 3(c) and 3(d), as well as in Figures 4 and 5, are not uniform. They also vary from figure to figure.] The data parallelism is concentrated in node 20 (outer product) and node 12 (membership). Most other parts are primarily sequential. Although PRIME does not use any floating-point instructions, for the sake of uniformity among our examples, the data shown were obtained with the data area declared noncacheable.

The second program, POISSON, solves a Poisson equation with boundary conditions on a rectangular grid, using the tensor product method. Results are shown in **Figure 4**.

The third program, MORGAN, is a time-consuming segment of code used in financial analysis. The main function has, among other operations, five calls to the subfunction

```
  ∇R←N MSUM A
[1] A←+\A
[2] R←((0,N-1)↓A)-0,(0,-N)↓A
```

where N is an integer and A is a 30 × 700 matrix of real numbers. **Figure 5** presents the results.

Even though we present our data for a fixed input size, both POISSON and MORGAN are compiled with the specification that input size is unknown at compile time in order to preserve the APL spirit of allowing dynamic arrays. Hence, any change of array size does not incur a recompilation. This has a slightly higher, but insignificant, run-time cost.

The data indicate that the best parallel speedup obtained for any of these examples is about 12. At first glance, this is a rather discouraging result for highly parallel machines. However, if we remember our simple run-time execution model and relate it to Amdahl's Law [26], it is not surprising at all. Amdahl's Law says, for example, that if 10% of an application is serial, the best parallel speedup that can be hoped for is ten, under the assumption that the time for the parallel part is 0. Our execution model certainly cannot circumvent the effect of Amdahl's Law, since we strictly split execution into serial and parallel parts. In fact, in [25] we have measured the execution time

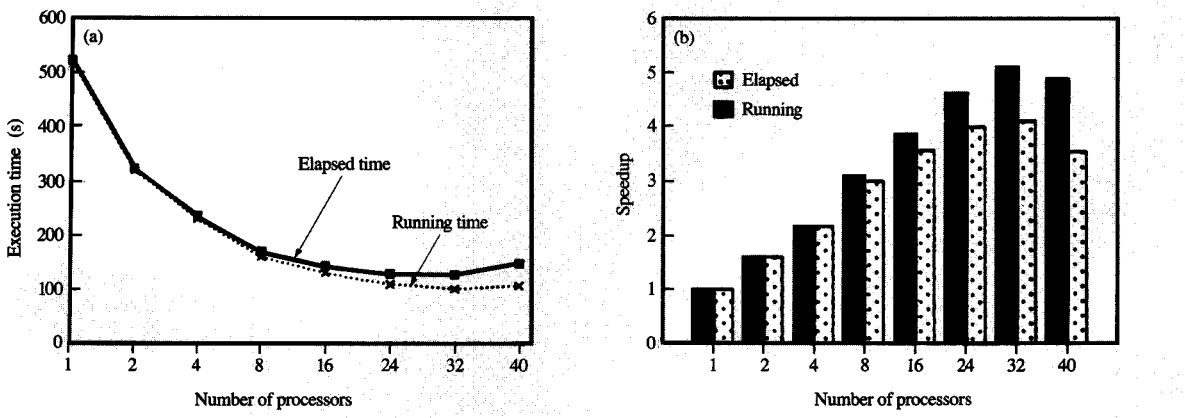


Figure 5

MORGAN program for a 30 × 700 matrix: (a) execution time; (b) speedup.

and speedup of parallel blocks only, and observed almost linear speedup, with up to 32 processors in some cases.

We give here several reasons why speedup peaks, even for parallel blocks. First, data access is much faster if the needed data are available in the cache. Because of the peculiar problem of floating-point-code generation by the C compiler, mentioned previously, and our subsequent declaration of data being noncacheable, most array data access cannot take advantage of the cache. Second, the arrays are laid out in the shared memory as they would be in a single address space for a uniprocessor. Hence, when executing a slice of a parallel block, a processor is likely to access memory that is not local. This means that the access must go through the network and incur overhead and delays due to network congestion. Finally, we did experiments for programs with a fixed input size. Thus, as more processors participate in executing a parallel block, each processor gets a small chunk of data to work on, while overhead to start a parallel block is fixed and network traffic increases because of increasing remote-access requests. These arguments explain much of the reason why the speedup is sublinear.

In any case, the bound on relative speedup also reveals the limitation of our current implementation—a strictly fork/join run-time execution model that is capable of exploiting data parallelism only. As pointed out in [11], our compiler front end can discover functional parallelism in all basic blocks of an APL program. It inserts synchronization flags to mark portions that can be executed in parallel with other portions up to a particular program node of the parse tree. For example, the two subtrees delineated by arcs in Figure 3(b) can be executed in parallel. More importantly, in many cases, the serial array-bound checking of one array operation can be carried out during the parallel computation of a preceding array operation. In short, a more sophisticated scheduling of fine-grain tasks and an elaborate run-time environment could help to minimize the serial bottleneck and achieve better speedup.

Nevertheless, we can improve our results without changing the run-time environment. The compiler front end groups sequences of consecutive scalar primitive functions into *streams*. However, the back end does not fuse the loops corresponding to these scalar functions automatically. Since fusing loops obviously reduces overhead, we did an experiment with the MORGAN example by hand-fusing loops after code was generated and achieved better speedup than that shown in Figure 5(b) (see [25]).

We also observed for all examples that when the number of processors approached 63 (RP3 reserves one processor for system use), the elapsed time actually increased slightly. This indicates that initialization cost, no matter how small, is not trivial for the case of a large number of processors. Also, when more processors go to work, each gets a smaller chunk of work, and unless the work-pile is very large, overheads incurred for more processors overshadow the gain from having more processors to do the work.

Now let us assume that we can exploit functional parallelism as well as data parallelism in each basic block of an APL program. On what kind of programs will our

automatic parallelization scheme be effective? Certainly not for programs that are literal translations of FORTRAN programs. A program must contain a large amount of array data parallelism to be able to benefit from the automatic parallelization that our compiler provides. In other words, a major part of the program must be expressible in terms of high-level array operations. That is what we call APL-style programs in [27]. We observed that APL-style programs for uniprocessors do not tend to run significantly faster when compiled and executed than when interpreted. We call this improvement of compilation over interpretation "sequential speedup." Sequential-style programs for uniprocessors, on the other hand, tend to display considerable sequential speedup. An interesting fact has emerged: The parallel speedup an APL program can achieve on a highly parallel machine such as the RP3 tends to be quite opposite to the sequential speedup of the same program [27]. That is, a good APL-style program that enjoys a moderate sequential speedup on a uniprocessor exhibits better parallel speedup than a sequential-style program, whose compiled version experiences great sequential speedup. Therefore, the programming style APL programmers developed to avoid inefficiencies in the interpreter is precisely what makes their programs naturally suitable for parallel execution. These conclusions, of course, apply to programs compiled with our compiler.

We note here that because our implementation uses primitive functions as units of parallelism, an interpreter could also exploit parallelism in a similar way (given enough resource and effort). But we remind the reader that even though our serial code is executed sequentially, it does not contain any interpreter overhead. For example, suppose we have a parallel machine that can execute the parallel portion of a program in no time. For a program P that spends one unit of time in interpretation, one unit in the sequential part, and eight units in the parallel part, the parallel speedup is five, since interpretation is mostly sequential. When P is compiled, the parallel speedup is nine, since there is no interpretative part. This is the effect of Amdahl's Law intensified by another factor (that of interpretation cost). Moreover, an interpreter cannot analyze data dependency to discover and exploit functional parallelism, which is crucial to lessen the effect of Amdahl's Law, as we have explained.

Conclusion

We have presented a description of an APL/C compiler that automatically parallelizes, at the APL primitive-function level, the C source code translated from an APL program and exploits data parallelism. Our run-time environment, built for running under Mach on the RP3 machine, is similar to but simpler than the PTRAN run-time environment. We use chunk scheduling and distributed locks for synchronization. We do not rely on profiling data to decide upon the number of processes for parallel execution. Speedup and timing data for several APL programs have been presented to show the relative effectiveness of our automatic parallelization. The reasons for sublinear speedup are as follows: Because our current implementation ignores functional parallelism, Amdahl's Law limits us; and data declared noncacheable and network traffic increase when more processors are used. We point out that the kind of programs that can benefit from the automatic parallelism our compiler can provide are the ones truly utilizing the very high-level nature of the array-oriented APL language. We also explain why interpreters are ill-equipped to exploit parallelism for overall speedup of applications.

Acknowledgments

We thank Frances Allen for her support of this project and for giving us good advice. We thank Paul Carini for teaching us the details of running code on the RP3 and the Mach operating system. He patiently spent many hours with us, to get us started and to help debug our programs. We thank Henry Chang for discussing with us his experience with RP3 and Mach-threads, to help us in trouble-shooting. We also thank Edith Schonberg for providing us with the source code used for the PTRAN run-time environment on RP3.

References

1. J. Clary, G. Howell, and S. Karmen, "Benchmark Calculations with an Unstructured Grid Flow Solver in a SIMD Computer," *Proceedings of Supercomputing '89*, Reno, NV, 1989, 32–41.
2. M. Kumar and Y. Baransky, "The GF-11 Parallel Computer: Programming and Performance," *Research Report RC-15494*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.
3. *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed., ACM Press, New York, 1991, Ch. 7.
4. *Ibid.*, Ch. 6.
5. *Ibid.*, Ch. 4.
6. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Dist. Computing* **5**, 617–640 (1988).
7. R. Cytron, D. Kuck, and A. Veidenbaum, "The Effect of Restructuring Compilers on Program Performance for High-Speed Computers," *Computer Phys. Commun.* **37**, 39–48 (1985).
8. M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis," *ACM Trans. Programming Lang. & Syst.* **12**, 341–395 (1990).

9. V. Sarkar, "Determining Average Program Execution Times and Their Variance," *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 1989, pp. 289–312.
10. G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Computer Society, 1985, pp. 764–771.
11. W.-M. Ching, "Automatic Parallelization of APL-Style Programs," *Proceedings of the APL'90 Conference*, Copenhagen, Denmark, 1990, pp. 76–80.
12. W.-M. Ching, "Program Analysis and Code Generation in an APL/370 Compiler," *IBM J. Res. Develop.* **30**, 594–602 (1986).
13. *Research Parallel Processor Prototype Principles of Operation*, Report No. RP3001-1, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.
14. S. F. Hummel and E. Schonberg, "Low-Overhead Scheduling of Nested Parallelism," *IBM J. Res. Develop.* **35**, 743–765 (1991, this issue).
15. R. Baron, *Mach Kernel Interface Manual*, Carnegie Mellon University, Pittsburgh, PA, September 1988.
16. T. Budd, "An APL Compiler for the UNIX Timesharing System," *Proceedings of the APL'83 Conference*, Washington, DC, 1983, pp. 205–209.
17. R. Bernecky, C. Brenner, S. Jaffe, and G. Moeckel, "ACORN, APL to C on Real Numbers," *Proceedings of the APL'90 Conference*, Copenhagen, Denmark, 1990, pp. 40–49.
18. A. Guillon, "An APL Compiler: the SOFREMI-AGL Compiler, a Tool to Produce Low-Cost Efficient Software," *Proceedings of the APL'87 Conference*, Dallas, TX, 1987, pp. 151–156.
19. G. C. Driscoll, Jr. and D. L. Orth, "Compiling APL: The Yorktown APL Translator," *IBM J. Res. Develop.* **30**, 583–593 (1986).
20. J. Brown, S. Pakin, and R. Polivka, *APL2 at a Glance*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
21. *Research Parallel Processor Prototype User Overview*, Report No. RP3021-1, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.
22. R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM J. Res. Develop.* **35**, 617–634 (1991, this issue).
23. R. Bryant, "The RP3 Parallel Computing Environment," *Proceedings of the USENIX 1988 Supercomputing Workshop*, Pittsburgh, PA, 1988, pp. 69–85.
24. W.-M. Ching, "Evon: An Extended von Neumann Model for Parallel Processing," *Proceedings of the 1986 Fall Joint Computer Conference*, Dallas, TX, November 1986, pp. 363–371.
25. D. Ju and W.-M. Ching, "Exploitation of Data Parallelism on Shared Memory MIMD Machine," *Proceedings of the ACM Symposium on Principles and Practice in Parallel Programming*, Williamsburg, VA, April 1991, pp. 61–72.
26. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conf. Proc.* **30**, 483–485 (1967).
27. W.-M. Ching, R. Nelson, and N. Shi, "An Empirical Study of the Performance of the APL370 Compiler," *Proceedings of the APL'89 Conference*, New York, 1989, pp. 151–156.

Received September 13, 1990; accepted for publication January 2, 1992

Wai-Mee Ching *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Ching received his Ph.D. in mathematics from the University of Toronto, Canada, in 1968. He taught mathematics at Louisiana State University, Fordham University, and Cornell University, and he was a Visiting Member at the Courant Institute of Mathematical Sciences, New York University. From 1977 to 1979, he was a senior member of the technical staff in Perkin-Elmer's Data Systems Division. Since 1979, he has been a Research Staff Member in the Computer Science Department at the IBM Thomas J. Watson Research Center.

Dz-ching Ju *Department of Electrical and Computer Engineering, University of Texas, Austin, Texas 78712.* Mr. Ju received his B.Sc. degree in electrical engineering from the National Taiwan University, Taiwan. He is currently a graduate student in the Department of Electrical and Computer Engineering at the University of Texas in Austin.