Int J Parallel Prog (2012) 40:514–531
DOI 10.1007/s10766-012-0197-6

Automatic Parallelization of Array-oriented Programs for a Multi-core Machine

Wai-Mee Ching · Da Zheng

Received: 23 December 2010 / Accepted: 5 May 2012 / Published online: 24 May 2012

Abstract We present the work on automatic parallelization of array-oriented programs for multi-core machines. Source programs written in standard APL are translated by a parallelizing APL-to-C compiler into parallelized C code, i.e. C mixed with OpenMP directives. We describe techniques such as virtual operations and data-partitioning used to effectively exploit parallelism structured around array-primitives. We present runtime performance data, showing the speedup of the resulting parallelized code, using different numbers of threads and different problem sizes, on a 4-core machine, for several examples.

1 Introduction

Hardware technology for building general-purpose parallel computers has advanced greatly during the past decade, resulting in parallel machines ranging from the supercomputers on the Top500 list (http://www.top500.org) having as many as 224,162 cores to multi-core desktop computers readily available at offices and schools. On the software side, the technology needed to effectively harness the computational power offered by the currently available parallel hardware remains a grand challenge. The research and development on how to program parallel machines is in a state of flux, as summarized in a recent article from UC Berkeley's ParLab [2]. One approach to programming parallel machines is to write parallel code using thread-programming,

W.-M. Ching (✉)
Department of Computer Science, Zhejiang Normal University, Jinhua 321004, Zhejiang, China
e-mail: waimee_ching@yahoo.com

D. Zheng
Department of Computer Science, Johns Hopkins University, Baltimore, MD, 21210, USA
e-mail: zhengda1936@gmail.com

open-source API OpenMP or MPI, but this is cumbersome and error-prone. Another approach consists of inventing new parallel programming languages for programming the parallel machines, but then one faces the challenge of adoption of these new programming languages by the scientific and programming communities. Our approach is part of a pragmatic yet ambitious effort to develop compiler technology of automatic parallelization (*autopar* for short)—to turn programs written in an existing language into working programs for parallel machines. Traditionally, *autopar* refers only to *sequential* programs that are analyzed to uncover parallelism not explicitly stated by the programmer, i.e. dusty-deck FORTRAN programs, as illustrated by the Polaris work [3]. We instead used "dusty-deck" APL programs where parallelism is evident but need not be explicitly stated by the programmer, and *autopar* here refers to transforming array-style programs into low-level C programs to be compiled by a C compiler with *autopar* capability [18]. While APL has inherently parallel semantics, how to actually realize this parallelism on a parallel machine is not trivial since one has to avoid inefficiencies of an interpreter as well as inefficiencies of naive mappings of array operations. The work on *autopar* which we present emphasizes how to generate efficient parallel code for combinations of array primitives on a multi-core machine. Thus, by carrying out detailed experimentation in implementing and measuring parallelism for a set of sample APL programs on an actual machine, we show that by utilizing very high-level array primitives in the source language one can effectively exploit parallelism of multi-core machines without going through the traditional route of extensive loop dependency analysis.

There are earlier papers on *autopar* of APL code by an APL compiler, on an experimental IBM 64-way shared memory machine RP3 [8], and on a distributed memory IBM SP-1 machine [11] exploiting data-parallelism inherent in APL primitives. The main focus in [8] is simply to build an experimental run-time environment based on the Mach operating system of RP3 for parallel execution of APL primitives. In our case, the runtime environment is provided by the OpenMP on an Intel-architecture 4-core machine running under Linux, and we focus on how to generate parallel C code with increased locality of reference and reduced data movement over code segments, to achieve good parallel result for modern multi-core machines. Most other work on *auto*-parallelism typically involves some additional modification to the source code by a user, such as data-layout declaration in HPF [13] or compiler directives for data partitioning. No such annotations are required in our case. However, for array-based autopar to be effective the source code needs to be array-oriented, i.e. it must use array primitives, instead of sequential loops, whenever possible. That is it must exhibit the *dataflow style* of programming—organizing tasks as chains of monadic or dyadic functions and operators acting on arrays—a style long practiced by APL users (*monadic* and *dyadic* refer to a function taking one or two arguments). This is because the parallelism we implement is inferred from array primitives, in contrast to the scalar language based approach where parallelism is reconstructed from loops. In theory, we can use FORTRAN90 to do this autopar experimentation, but in practice there are difficulties, which we will discuss later in Sect. 5. Another reason we chose APL is its succinctness of syntax and symbols, yielding a compiler which is easily modifiable for autopar experiments.

We started with a base APL-to-C compiler [10], *COMPC*, i.e. a translator generating C code from APL programs. We then experimented with parallelizing the generated C code under Linux, by inserting light-weight synchronization directives provided by OpenMP [20]. Next, we wrote a parallelizing module, *PARAC*, and integrated it into the base compiler, to reorganize the generated C and insert additional parallel directives required for low-level synchronization. A good portion of the work in the *PARAC* module was inspired by examining the performance of programming patterns found in the sample programs we used, and which we concluded can benefit APL programs in general. The main techniques incorporated into *PARAC* to enhance parallel performance are ***virtual operations*** and ***data-partitioning***; the first attempts to reduce the number of data operations when implementing some combinations of array primitives in certain contexts, while the second aims to increase locality of reference. The parallelizing module and the use of OpenMP are based on the assumption that the generated C code will run on a shared-memory machine, i.e. a current day multi-core machine.

We measured the run time and speedups of several sample programs. These sample programs are not large, but neither are they just simple linear algebra routines. They are not massively parallel applications. Other than example 1, they come from real scientific and financial applications. The speedup achieved is not linear yet reasonable, and considering that the speedup comes at no cost to the programmer or the user, this benefit of utilizing the power of a multi-core machine is quite attractive. The methods we developed here are not specific to APL. Rather, they point to a promising approach to auto-parallelism—to take advantage of the increased level of parallelism in the primary data structures used in a program, whether array-oriented or object-oriented, coupled with a supporting parallel implementation.

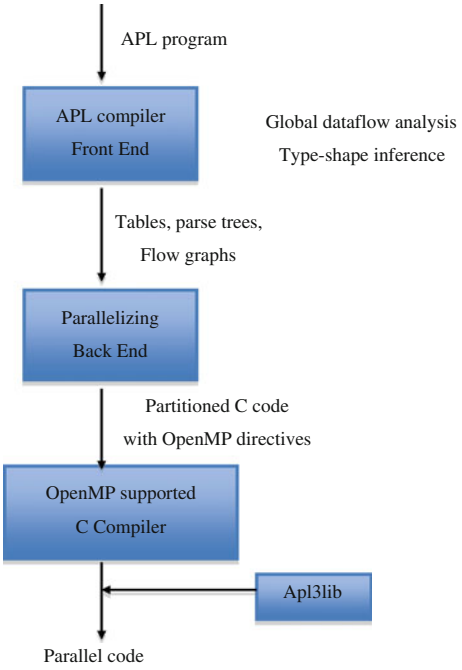
In Sect. 2, we summarize the basic features of the base APL compiler and the overall framework for producing parallelized C code by judiciously inserting OpenMP directives to implement fine-grain parallelism. In Sect. 3, we describe techniques we developed to improve parallel performance of the generated C code. In Sect. 4, we present the measurement of run times and speedups of several examples, some specific compiler work related to each example and the impact of the cache on the speedup. In Sect. 5, we discuss the effectiveness of our approach, and future extension, FORTRAN90's limitation, and suggest a programming style that is suitable for autopar of C++ programs. Finally, after the conclusion, we include in the appendix an explanation of the APL primitives used in the sample APL programs.

2 The Base Compiler and the Parallelizing Back End

APL was invented by Ken Iverson at Harvard for teaching applied mathematics and further developed at IBM T. J. Watson Research Center. For his work on APL Ken Iverson received the Turing Award in 1979 [14]. A line of APL code executes from *right to left*, all functions have *equal precedence*, and operators apply to primitive functions. A symbol can represent either a monadic or a dyadic primitive function, depending on whether it takes one or two arguments. Since an APL execution environment is usually implemented by an interpreter, APL programs execute much slower

than corresponding C programs. For this reason, APL was not considered suitable for computationally intensive jobs and largely ignored by the parallel programming community.

In the 1980s, APL was extended to include nested arrays, but we restrict ourselves to classical APL (i.e. flat arrays) as described in the ISO Standard [19]. Most of the classical APL can be compiled with resulting efficiency comparable to that of FORTRAN programs (see performance comparisons in [7]). The base APL compiler [10], *COMPC*, translates APL into C. *COMPC* is written in 14K lines of APL with a small C runtime library linked in with the C code produced by the *COMPC* translator, to create the executables (see the figure below). Without the inefficiency of the interpreter, APL becomes a quite attractive research vehicle for studying the effectiveness of various proposed compilation techniques aimed at exploiting parallelism implicit in array-oriented language primitives. Unlike new languages explicitly designed for parallelism, APL has been widely used for many years with a large body of array-oriented programs on a wide range of applications (such as chip design, logistics, financial applications, etc.).



COMPC accepts an unmodified (classical) APL program and produces a corresponding ANSI C program. The main restriction on the input APL programs, with respect to ISO standard in [19], is the exclusion of the execute function $\underline{\phi}$. No variable declaration is required, but when invoking the compiler the user needs to specify the type and rank of input parameter(s) to the top-level function. In general, compiled APL programs execute about 10 times faster than the same programs running under the interpreter on an IBM workstation (see [10]).

The basic features of the *COMPC* are:

1. The parser is not *yacc* like, instead it uses a left-to-right two symbols look-ahead algorithm [6].
2. The front end implements the full machinery of the Allen–Cocke interval-based dataflow analysis [1] not for optimization, but for deducing type/shapes of all variables from the input parameter(s).
3. The optimization strategy is primitive (function)-based [9]. The main function in the back end, *TREELIST,* walks through a parse tree from the lower right node up to the root in a semi-recursive fashion and for each internal node calls various *PF*-functions. PF-function stands for *primitive function*, i.e. for each APL primitive/defined function encountered *COMPC* calls the corresponding code-generating *PF*-function.

To reduce unnecessary temporary variables *COMPC* uses a simple algorithm to find all *stream*s, i.e. a group of consecutive scalar functions (arithmetic, logical, and relational function) nodes, where the output of one feeds the input of another, in a parse tree of a line. The back end then generates one fused loop, corresponding to the classical loop-fusion for FORTRAN programs. For example, $A \leftarrow B + C \leftarrow D \times E$ is a *stream,* where the product $D \times E$ is assigned to C and added to B to produce A [6]. Primitive functions in APL are called scalar when they operate on same-shaped arguments by applying the function to each pair of corresponding elements. All arithmetic functions like $+$, $-$, $\times$, etc., are scalar, since they can be applied to two same-shaped arrays A and B, as in $A + B$, $A \times B$, etc., by applying the $+$ or $\times$ function to the corresponding pairs of individual (scalar) elements of the A and B arrays, regardless of the dimensions of the arrays, as long as the two arrays have the same number of dimensions, and the same number of elements in each dimension (same rank and shape in array parlance). Similarly all logical (and, or, nor, etc.) and relational ($<$, $>$, etc.) functions are scalar functions. All arithmetic dyadic scalar functions also have a monadic scalar function version—for example there is the dyadic $A - B$ and the monadic $-A$. For the monadic scalar functions the arithmetic function is also applied to each element of the argument array, independently of all other elements. It is a sequence of such consecutive scalar functions which we call a *stream*, and which our compiler optimizes.

Non-scalar functions in APL are called *mixed functions*. Examples of mixed functions include membership, index of, rotate, etc. For mixed functions each argument's shape and rank can (or must) differ from the shape and rank of the other argument, and the function is not applied element by element to the two array arguments. For example membership ($A \in B$) returns a boolean array of the same rank and shape as A, with a 1 for every element of A which exists in B, and a 0 for every element in A which does not exist in B. It is the consideration of every element of B, when producing a result for a single element of A, which makes membership a mixed function.

We set as our objective the automatic restructuring of the translated C code, through a parallelizing compiler back end, with added OpenMP directives, to make the resulting program execute efficiently in parallel, without the need for the end user to modify the source APL program. OpenMP is a very popular API for writing parallel applications in C/C++ and FORTRAN, for shared-memory parallel machines. As OpenMP

is supported by all major hardware and software vendors, applications using OpenMP are quite portable across a wide variety of hardware, making OpenMP an attractive platform for our research. In particular, *gcc* supports OpenMP [18]. Using OpenMP has a clear advantage over hand-threaded approaches to parallelism such as using Pthreads because using OpenMP eliminates the need to create and manage individual threads manually. Instead, one only needs to insert directives into the sequential code to tell the OpenMP compiler where the parallel portions reside inside the sequential code, to execute in parallel by multi-threads on a multiprocessor. However, to achieve good performance and scalability, one needs to identify the right portions of the code for parallelization, inserts appropriate directives into the code, partition the data in a proper manner, and sometimes one even needs to rewrite some portions of the sequential code. So to manually parallelize an existing C/C++ code, using OpenMP, is no simple matter and requires a thorough understanding of the sequential code. In short, while OpenMP provides convenience and portability for parallel programming, it does not provide *automatic* parallelization.

Most work on automatic parallelization, if not based on a new parallel language, assumes the source code comes from a sequential language such as C/C++ or FORTRAN, and uses compilation techniques to free the developers from having to:

- find loops that are good work-sharing candidates,
- perform the dependence analysis to verify the correctness of parallel execution,
- partition the data for threaded code generation as is needed in programming with OpenMP directives.

The source-to-source transformation by the Cetus compiler infrastructure [4] gives a good example of the extensive analysis involved.

In contrast, our approach assumes that the original source code is written in an array language which uses array primitives, not loops, to perform calculations, and manipulates arrays, whenever possible. Hence, even though the source to be parallelized with OpenMP is C, we know that it came from APL and we know where and how to effectively put OpenMP directives without extensive loop analysis. For example, we know which loops perform *reduction* as our base compiler already takes note of this in its front-end analysis.

Our modification to the base compiler goes beyond inserting OpenMP directives intelligently—more importantly we need to restructure the translated C code to achieve good parallel performance. This task was neither straightforward nor excessively complicated. Most modifications were to the code generating back end of the base compiler. The base compiler *COMPC* is an APL workspace consisting of 307 functions. The module for parallelization added to the base compiler is a smaller APL workspace named *PARAC* consisting of 37 functions (and 2,461 lines of APL code). When the *PARAC* workspace is copied, after loading *COMPC*, the resulting workspace has 324 functions and 14,560 lines of code (many functions in *PARAC* replaced old functions in *COMPC*). Hence the module results in 17 new functions and 610 lines of new code, and it does add a bit to the compilation time (for the PRIME example in Sect. 4, the uniprocessor C version is generated in 0.016 vs. 0.032 s for the parallel version). The *PF*-functions modified by *PARAC* include *PFSCALAR, PFINNERP, PFOUTERP, PFROTATE, PFDROP, PFCATENA*, which respectively generate code for scalar

functions, inner product, outer product, rotate, drop and catenate. For example, the following is a segment of generated C code for outer-product ∘.× in the PRIME example:

```
 /* PFOUTERP generates the following code */
lo1 = ( int *) v25.valp; ro1 = ( int *) v23.valp;
 cad[2]= v25.reall; cad[3]= v23.reall;
 r0 = v25.reall*v23.reall; cad[1]=r0;
#pragma omp parallel for default(shared) private(v1, v2)
 for (v1=0; v1<v25.reall; v1++)
  for (v2=0; v2<v23.reall; v2++)
   v5[v1*v23.reall+v2] = lo1[v1]*ro1[v2];
```

As we described earlier, a *stream* is a group of consecutive scalar function nodes where the output of one scalar function feeds the input of another. The back end attempts to generate one fused loop as in classical loop-fusion for FORTRAN/C programs. We use the term *streaming* to refer to the process of generating a fused loop for the execution of a stream of scalar primitives. *COMPC* has two ad hoc restrictions in forming a stream: no boolean operand and no left sub-tree operand (expressions inside parentheses) can occur in a stream. In *PARAC,* we removed these two restrictions so streams with boolean operands and left sub-tree operands can be fused by the back end now. To do this we added a new way to access boolean variables (*BOPERAND* in *PARAC*), since *COMPC* packs 8 boolean values into one byte thus breaking a stream. *PARAC* also modified two functions *TREEWALK* and *QSTREAM* in *COMPC*. As a result for the stream (line 7 in **jacobi** Sect. 4) `A−Z←C+0.25×F×R4+R3+R2+R1` (note there is no operator precedence in APL: from right to left the sum of R_i is multiplied by the boolean variable `F`, then multiplied by `0.25`, added to `C`, and assigned to `Z`; the result in `Z` is subtracted from `A`), the compiler generates a compact C code loop:

```
for (v1 =0; v1<r0; v1++){
   p20[v1]=lo22[v1]+0.25*BVAL(v19,v1)*(lo21[v1]+(lo20[v1]+(lo2[v1]+ro2[v1])));
   p2[v1] = fabs((lo23[v1]-p20[v1]));
}
```

In addition to the new and modified functions we introduced in the workspace *PARAC*, we also introduced new macros in Aplc.h and functions in Apl3lib.c to make it easier to generate parallel C code. We further parallelized some functions in the library Apl3lib, which comes with *COMPC*.

3 Virtual Operations and Data-Partitioning

For multi-core computers, parallel performance greatly depends on keeping the needed data in the cache as long as possible, since any swapping of relevant data in-and-out of the cache, during computation, will drastically degrade performance. This is similar to trying to keep the most frequently used data items in registers in classical compiler optimization, and loop fusion we described above certainly serves this purpose.

The technique we called ***virtual operations*** *(v-ops)* is also a way of combining array operations; but unlike streaming it combines an array transformation primitive with other primitives. It is similar in spirit to the delayed evaluation scheme outlined

in Guibas and Wyatt [12]. We note parenthetically that the recent work of Liu et al. [17] is a comprehensive study of optimizing array computations, but it considers general loops. The idea of *v-ops* is to eliminate avoidable memory copy operations for APL primitives which rearrange the contents or alter the shape of an array. For example, when a *rotate*, or *drop*, on an array A is followed by calculations involving A, instead of carrying out the rotate and following it with the computational operation, the compiler *virtualizes* the rotate by a transformation of array indices for the next computation, to produce a semantically equivalent result (see example 3 in Sect. 4). We call the result of a *v-op* a *pseudo array* which occupies the same memory location as the *original array* (input to the *v-op*). At present, we only implemented *v-op*s for *drop*, *rotate* and *catenate*, but the technique we used can be applied to the *take* and *indexing* APL primitives as well.

To implement *v-ops*, we introduced a list `PSUEDOV` in the compiler to indicate which variable is *pseudo*, and if so what v-op applies to it. `PSUEDOV[10]<0` means `v10` is a normal variable; `PSUEDOV[11]=4` means `v11` is a pseudo variable as the result of a *virtual drop* (primitive function number `4`). At the entry to a basic block, `PSUEDOV` is initialized to all `-1`. Later the compiler calls `ISPSUEDOV` to check whether a variable is a *psuedo* variable. To generate the correct code for *v-ops*, we added two fields in the `struct` representing the shape of a variable which is an array of known rank but undetermined dimensions (see sec. 4 in [6]):

```
union pseudo_u *pseudo;
int flags;
```

where union `pseudo_u` stores additional shape information for calculating indices. `pseudo_u` has two structures: `moref` and `pinfo: pinfo` is used by the original array to keep track all pseudo arrays that point to it; `moref` stores information about a pseudo array using three 3 fields: `s_startp` stores the starting location of each dimension in the original array, `orig` points to the original array and `data_avail` is used by data partitioning.

When a pseudo array `pa`, based on original array `a`, is accessed, the compiler chooses a corresponding macro to generate code. Basically, references to array elements `pa[i;j]` are transformed to references to array elements in `a`, `a[f(i);g(j)]`, where functions `f` and `g` depend on the *v-op* involved. For example, to access a pseudo floating-point matrix rotated along the last dimension by 1, the macro `LROTATED_DVAL2` is called (The names of these macros are derived using the following rule: the first part (before "_") indicates the *v op*, `D` indicates the data type of the array (double), and `2` indicates the number of dimensions). In this case f is the identity function, and g is `leng2|j+1`, where `leng2` is the length of the last dimension of `a` as the compiler uses index origin 0 as in C. f and g for *drop* is very similar to rotate while they are more involved for the *catenate* primitive since it glues two arrays.

The other technique we used for effective parallelization is ***data partitioning***: we partition an input array into groups and process each group in one core through all the code execution corresponding to a sequence of APL primitives. Once the proper size is determined, each core only needs to load its partition for the entire sequence of primitive functions. This can be considered an extension of streaming, but instead of

processing one element through a stream of primitive functions, we process a portion of the data. Moreover, unlike in streaming, data partitioning can include mixed functions (i.e. those that do change the shape of an array) like rotate, take, drop etc. To increase the effectiveness, we allow data partitioning to be mixed with virtual operations. This complicated the implementation a bit, since one partition may rely on another partition which is processed in another iteration or even in another core. Hence before executing each function on a partition, the generated code checks the data dependency of partitions at run time. To implement this checking the compiler generates macro wrappers around the generated C code. Before processing a partition, the generated code calls `NEXT_CHUNK` to find the next partition to process. Then it calls `DATA_AVAIL()` to check whether the partitions, on which this next partition relies, are available. After finishing processing, the code calls `UPDATE_RANGE()` to mark that the current partition has been processed. We call the execution of one sequence from `NEXT_CHUNK` through `UPDATE_RANGE()` an *iteration*. If any of the depended-on partitions are not yet available, the execution jumps to the next primitive in the sequence, instead of blocking and waiting for the data to become available, i.e. in each iteration, only the partitions for which depended data is already available are processed. This guarantees that there is no dead lock in the generated code. We record information of all processed partitions in records describing variables. Thus each variable can not be assigned multiple times.

We illustrate the above discussion with the C code generated for the APL source code `out←in1+in2`, where `in1` and `in2` are two floating-point matrices, with the data partition loop, where the OpenMP directive is only used to create threads while `DATA_PART_LOOP` is responsible for distributing data to each thread and initializing the current thread for data partitioning:

```
#pragma omp parallel for default(shared) private(thread_num)
DATA_PART_THREAD_LOOP
DATA_PART_LOOP {
 ITERATION_START;
 NEW_ASSIGN(out); NEXT_CHUNK(out);
 if (DATA_AVAIL(in1)&&DATA_AVAIL(in2)) {
   FORALL2_CACHE(out)
   DVAL2(out)=DVAL2(in1)+DVAL2(in2);
   UPDATE_RANGE(out);}
}DATA_PART_THREAD_LOOP_END
```

4 Parallel Speedups of Sample APL Programs

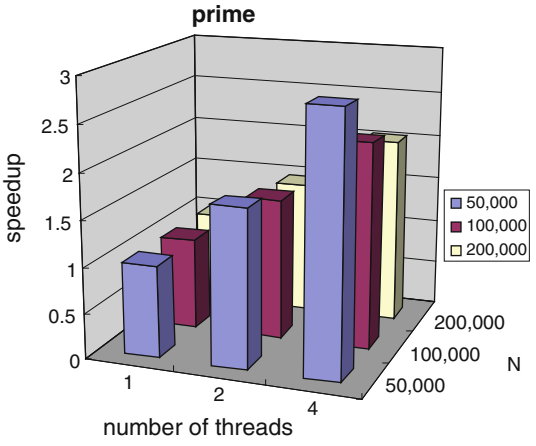
We report the performance of our parallelizing APL compiler for four moderate-size examples, together with indications of different techniques deployed to speed up these programs. The multi-core computer we used is an AMD Athlon II X4 620, clocked at 2.6 GHz and with 4 × 512 KB L2 cache. The computer was equipped with 2GB DDR3-1333 memory with good access speed. The desktop computer ran Debian/Linux 5.0. The C compiler we used to generate machine code is **gcc** v4.4 for Linux, which supports OpenMP 3.0. We attached one thread to each CPU and ran the

compiled programs with a higher priority to reduce the interference from other processes with our program's execution. We ran each program at least 5 times to get the average timing. We also measured the runtimes of these C programs using Intel's **icc** compiler, which has interesting variations but similar speedups (the timings are not listed here). We note that in the tables below the timing for 1 thread for **prime** and **poisson** is the time for running C code produced by the base compiler, while **timing** for **jacobi** and **morgan** is from the C code produced by the compiler with the parallelizing module. We did this because the parallelized C code for **jacobi** and **morgan** is very different from the code produced by the base compiler (*COMPC*) and the *PARAC* version is significantly faster than the original *COMPC* version, even when run in a single thread.

The first example is pedagogical: finding primes up to the integer N, using a single line of APL code. The APL code starts by building a vector V of odd numbers starting from 3, on the right side of the reduce operator ('/'). ⍳⌊(N-1)÷2 generates the integer list from 1 to about half N. The expression then builds a multiplication table of suitable size (the left argument) using an outer product (∘.×) of two integer lists with the first 2 integers dropped. Finally, those numbers in V which are not in the multiplication table are primes. The integer 2 is pre-pended to the vector of calculated primes, since 2 is prime but the vector V and the generated multiplication table contain odd numbers only. Two primitive array operations in this line of APL are suitable for parallelization: the outer product and the membership. The outer product is parallelized by the compiler by inserting an OpenMP directive, while the membership function is parallelized directly in the Apl3lib.c library, through a newer parallelized implementation of the membership function.

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

PRIME < 100,000	Time (ms)	Speedup	Outer product (ms)	Speedup	Membership (ms)	Speedup
1 thread	216.8	1	59.4	1	157.2	1
2 thread	140.8	1.54	30	1.98	110.4	1.42
4 thread	96.6	2.24	17	3.50	79.2	1.98

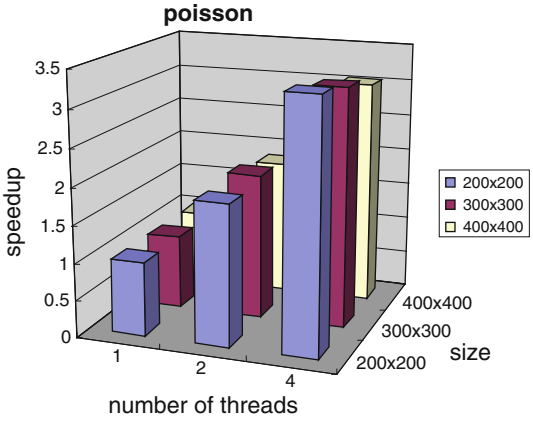


We see that outer product (∘.×) has an almost linear speedup while membership (∈), with its implementation by a library C function *membspi*, does not show very good speedup. *membspi* searches for elements of the left argument in the right argument and it is implemented with a hash table. Searches using a hash table are not easy to parallelize, and we only did the minimum amount of work to hand parallelize the *membspi* code. We need to further study the pattern and behavior of searching, and come up with some innovative implementation approaches to better parallelize *membspi*. We measured the performance of this prime generator with different input sizes for N: 50,000, 100,000 and 200,000 and show the results in the diagram above.

The second example is **poisson**, which solves the Poisson equation with boundary condition. Here we achieved a good speedup for all 4 inner products +.×, i.e. the matrix multiplications on line 12.

```
[0] Z←POISSON RMINBU;P;Q;L;M;S;T;V
[1] Z←0 0⍴0          ⍝ set empty matrix
[2] →(2≠⍴⍴RMINBU)/0 ⍝ if not 2 dims exit
[3] P←1+¯1↑⍴RMINBU ⍝ 1+2nd dim of input matrix
[4] Q←1+1↑⍴RMINBU ⍝ 1+1st dim of input matrix
[5] L←¯4×(1○∘.×(⍳Q-1)÷2×Q)*2 ⍝ 1○ is sin fn, ∘.× is pi×v
[6] M←¯4×(1○∘.×(⍳P-1)÷2×P)*2
[7] S←1○∘.×(⍳Q-1)÷Q
[8] S←S÷(+/S[1;]*2)*0.5
[9] T←1○∘.×(⍳P-1)÷P
[10] T←T÷(+/T[1;]*2)*0.5
[11] V←L+.×M
[12] Z←S+.×((S+.×RMINBU+.×T)÷V)+.×T
```

POISSON (300 × 300)	Time (ms)	Speedup	Inner prod (ms)	Speedup
1 thread	432.8	1	113.8	1
2 thread	220.75	1.96	52.25	2.18
4 thread	134.75	3.21	30.5	3.73



The total runtime, for a 300 × 300 input matrix, and the time for one inner product, are shown in the table above. We note that inner product achieves superlinear speedup

on 2 threads. We also measured the performance of **poisson** with different size inputs: 200×200, 300×300 and 400×400 with the results shown in the diagram above. For the **poisson** example, we noticed that the Intel compiler exhibited better performance. The main reason is that the Intel compiler automatically vectorized many C loop operations (http://software.intel.com/sites/product). Both **prime** and **poisson** have no loop in their APL source.

The third example is **jacobi**, which iteratively computes temperatures of interior points from those at the edges of a rectangle, by averaging their neighbors temperatures ($R1$–$R4$). The left argument `F` is a boolean matrix with 0s on the boundary and 1s at interior, right argument `A` is a floating point matrix denoting initial temperatures with interior points all 0. Here both streaming and virtual operations are crucial to achieving optimized parallel result.

```
[0] Z←F JACOBI A;C;E
[1] E←0.1
[2] C←(Z←A)×~F ⍝ Z,C get A as initial value
[3] L:R1←¯1⊖A←Z ⍝ upper rotate A by 1 row
[4] R2←1⊖A ⍝ down rotate A by 1 row
[5] R3←1⌽A ⍝ right rotate A by 1 column
[6] R4←¯1⌽A ⍝ left rotate A by 1 column
[7] ←(E<⌈/|,A-Z←C+0.25×F×R4+R3+R2+R1)/L ⍝ taking ave-
    rage and diff
```

The result shown below is with the optimization of merging scalar primitives only, i.e. only using streaming:

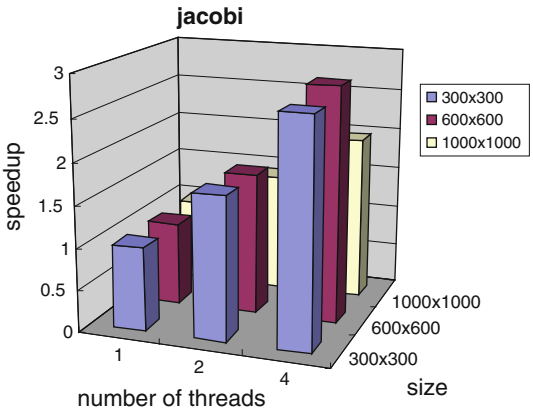
JACOBI ($1,000 \times 1,000$)	Time (ms)	Speedup
1 thread	7,174	1
2 threads	6,031	1.19
4 threads	5,937	1.21

Initially, we found almost no speedup when running on 4 threads compared to running on 2 threads. The main reason is that whenever the program executes the rotate operation, it generates a new array. Hence, the program has to load many arrays during each iteration. Since these rotated arrays are read-only in this program, we can avoid doing a real rotation. That is, by applying a *virtual* rotation, no new array needs to be created to carry out the rotation operation. Instead, the indices are calculated to access the array, at the position where the element ends up after the rotation (the index-calculating code is generated by the *PFROTATE* function in *PARAC*). With virtual rotation, the performance, as shown in the table below, has a very good speedup on 4 threads.

JACOBI (600×600)	Time (ms)	Speedup	In stream (ms)	Speedup	Reduction (ms)	Speedup
1 thread	3,046	1	2,587	1	188.2	1
2 thread	1,791	1.7	1,398	1.85	146	1.29
4 thread	1,082	2.8	705.4	3.67	105.8	1.78

The rest of time in **jacobi** is consumed by the memory copy in the loop, but outside of the stream. We noticed that the Intel compiler almost always produced a better result except, for the stream running on 4 threads. By inspecting the assembly code,

we realized that the code segment for stream and reduction is vectorized automatically by the icc compiler, with the resulting code running much faster.



We also measured the performance of **jacobi** with different size inputs and we illustrate the results in the figure above. We see from the graph that **jacobi** has significantly worse speedup with the input size of a $1,000 \times 1,000$ matrix running on 4 threads. Execution on 2 thread still has quite a good speedup, but 4 threads have only 23 % speedup over 2 threads. The speedup of the streaming operation running on 2 threads is 72.6 % and only 40 % on 4 threads, because the memory bandwidth becomes the bottleneck. We estimate the memory bandwidth according to this result: the execution of the stream in the last line of the APL code takes 8ms on 4 threads. Because the arrays are very large, all data has to be loaded from the memory and the stream needs to read or write 8*6 MB data. Suppose that all memory pre-fetch is accurate, the memory access rate is 8*6/0.008 = 6,000 MB/s. The memory bandwidth we measured with some professional software is about 7,000 MB/s. Since we need to calculate indices to access each element in the rotated matrix, this takes a considerable amount of time, so we believe we have reached the bottleneck of memory access in this example. Unless we adopt a different way to take advantage of the cache (in our current code, a bigger cache cannot help the performance), it is difficult to make the program run faster.

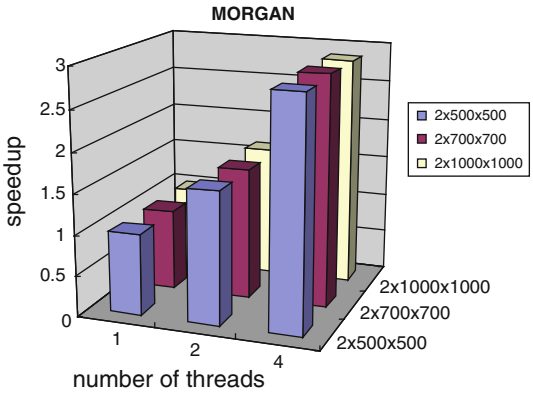
The last example **morgan** is extracted from a real-life financial application. The APL source has two user-defined functions. Since parameter passing in an APL function call is **call-by-value**, we implemented a limited *in-line expansion* capability in the compiler front-end to avoid huge data copying in the 5 function calls to *MSUM*. The whole calculation in the last line of **morgan** is done in a single stream. Hence, allowing left operands to be subtrees in a stream made substantial contribution to the performance improvement. Currently, data partitioning only works on a sequence of operations on arrays of the same rank. In the case of **morgan**, lines 3–8 are in a range of one data partitioning. We point out that data partitioning plays a crucial role in achieving good parallel performance in this last example. The speedups are shown in

the diagram below and the details for the middle size input data are in a table above that diagram.

```
[0] R←N MORGAN A;X;Y;SX;SX2;SY;SY2;SXY
[1] X←A[1;;]
[2] Y←A[2;;]
[3] SX←N MSUM X
[4] SY←N MSUM Y
[5] SX2←N MSUM X*2
[6] SY2←N MSUM Y*2
[7] SXY←N MSUM X×Y
[8] R←((SXY÷N)-(SX×SY)÷N*2)÷(|((SX2÷N)+(SX÷N)*2)*0.5)×
    (|(SY2÷N)-(SY÷N)*2)*0.5
```

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

MORGAN ($2 \times 700 \times 700$)	Time (ms)	Speedup	Data partitioning (ms)	Speedup
1 thread	115.2	1	108.2	1
2 thread	70.8	1.6	64.6	1.67
4 thread	40.2	2.9	34.2	3.16



In summary, we see that all the sample programs always have quite good speedup running on 2 threads, but not necessarily on 4 threads. This is understandable because the memory bandwidth for one CPU is the same, no matter whether we use 1 core, 2 cores or 4 cores. But the bottleneck due to memory speed becomes more and more pronounced when we use more cores. We can have better performance either with faster memory access or bigger cache. The four examples show the challenges in programming for the multi-core architecture: how to adjust emitted code properly so that the needed data can reside in the L2 cache as long as possible, and how to avoid naive implementation of carrying out all array operations one at a time. Indeed, one problem with array-oriented programs is that a strict interpretation frequently incurs a large number of avoidable operations resulting in longer execution times.

Though the examples chosen here are of only moderate size, they do cover a wide range of features in APL programs, including iterations and function calls. Thus the parallelizing techniques we implemented do apply to typical APL programs, involving array primitives and frequently used operators.

5 Discussion

People familiar with APL may suggest another approach to attaining parallel execution of APL: using the shared variables facility of the interpreter, together with the ***each*** operator in APL2. However, without removing the inefficiency of interpretation, achieving good performance is problematic. This approach would also incur the overhead of using the shared variables facility, which adds to the execution code path. In any case it would require a user to reorganize the program around the use of the ***each*** operator to explicitly express parallelism in the application.

The general ideas of data partitioning, interchanging and merging is well known, and presumably we could carry out our experiment in FORTRAN90 or HPF. Indeed, there is a study of parallelizing array language primitives on an early shared-memory machine using examples from LINPACK recoded in FORTRAN90 by Ju et al. [15]. However, we note that this is done in a language with variable declaration and data layout indication to the compiler, neither of which is needed in our approach. In other words, while FORTRAN90 and its compiler expect some cognizance from the programmer that the program will run in a parallel execution environment, our work on parallelizing the execution of APL programs allows the APL programmer to remain oblivious to the execution environment. Our goal in this research is to show that we can achieve significant speed up parallelizing array-oriented source code, without imposing on the source code programmer the need to consider anything other than the application logic of their program. The APL programmer needs not care whether the program will run under an APL interpreter, or be executed on a multi-core after compilation by our parallelizing compiler. After all, the main job for the APL programmer is to write good APL code. Furthermore, while FORTRAN90 has many array operations borrowed from APL, the set of array operations is not nearly as rich as in APL. A number of APL array operations, such as membership and drop, are missing entirely from FORTRAN90. We demonstrated a detailed yet elegant scheme for parallelizing array-based programs, without the need for variable declarations or data layout compiler directives.

From the performance data listed in the previous section, we see that the effectiveness of our approach to automatic parallelization depends critically on a program being written in an array-oriented fashion, i.e. on using array primitives whenever possible. Just as in a typical computation intensive FORTRAN or C program most computation time is spent in loops, in an array-oriented program most computation time is spent in array primitives. We achieved data parallelism by parallelizing array operations in an integrated fashion by the compiler, without any knowledge of the structure of the application involved, nor did we do any elaborate scalar-loop analysis. The additional compiler work of partitioning arrays and optimizing the use of temporary variables,

introduced in the data partitioning process, can further reduce the cache miss ratio and improve performance.

To achieve automatic parallelism on a wider range of applications, we need to implement task parallelism in addition to data parallelism. This would involve doing dependency analysis to identify independent tasks in basic blocks, and a more application specific coarse-grained analysis of programs (for massively parallel machines). Nevertheless, in all these efforts, having a source program written in a very high-level language still has a distinct advantage: the usage of very high-level primitives naturally enlarges basic blocks and reduces the complexity of flow graphs for intra-procedural analysis and call graphs for inter-procedural analysis. To extend our work in another direction, i.e. to cover distributed memory or hybrid memory parallel machines such as Cell processors or CUDA, we need data distribution analysis as done in [11], use MPI instead of OpenMP, or a more specific underlying support infrastructure, e.g. a parallel environment/library for executing appropriately parallelized code.

The automatic parallelization approach presented here is not restricted to APL. A similar effort can be applied to other very high-level array languages such as MATLAB and its open source siblings. Interestingly, from a recent comparison study [5], we see that MATLAB currently offers a two-pronged approach to extracting parallelism: an automated approach of implementing highly optimized and parallelized runtime routines, such as matrix multiplication, for the interpreter to call, and a non-automated approach of a set of tools available to the user to parallelize the code manually [21]. Neither approach involves compiler manipulation, thus differing from our approach. The recent work of Li et al. [16] on the scripting array language R is also on parallelizing its run-time routines but it did use sophisticated compiler technology to do this.

The demonstrated success of our approach to automatic parallelism described here can even suggest a *restricted* way of writing C++ programs for *auto*-parallelization, i.e. a programmer uses functions in STL instead of loops whenever possible, coupled with an enriched C++ compiler which automatically includes OpenMP directives in its implementation of array handling functions in STL (an effort along this line has apparently been initiated by a research team including Stroustrup, the designer of C++, at Texas A&M University, see http://parasol.tamu.edu/stapl/). While such a restriction may initially cause inconvenience to programmers used to writing scalar loops, the improved performance is still a bargain compared to the effort required to write hand-crafted parallel code or to write in a completely new parallel language to harness the parallel power built into today's multi-core machines.

6 Conclusion

We presented a work on automatic parallelization of array-oriented programs for desktop multi-core machines. The source programs were not modified for parallelization, but written in an array-oriented style. The parallelization was done by a parallelizing APL compiler which is based on an APL-to-C translator. The parallelizing module not only inserts OpenMP directives to the C program for parallel execution under

Linux but also restructures the emitted code, using the techniques of streaming, virtual operations and data partitioning, to achieve optimal results. Good speedups, which come at no cost to the user other than the need for compilation, were observed in our sample programs, typical of APL applications. We also discussed issues such as cache miss ratios and memory access with respect to the speedup we observed, and possible enhancements to the compiler. Our work shows that by writing in a very high-level (array-oriented) programming style one can effectively utilize the parallelism inherent in current multi-core machines through a parallelizing compiler of moderate complexity, without any explicit awareness of a parallel programming model or a need to include in some parallel abstractions in the design of the program.

Acknowledgments We thank Dr. Alex Katz for his help in editing the manuscript to improve its English. We also thank referees for their help in improving this paper.

Appendix

See Table 1.

Table 1 APL notations used in the sample programs

Name	Symbol	Function
Interval	⍳R	Generate ascending integers, i.e. ⍳4 generates a vector 1 2 3 4
Catenate	L, R	Concatenate two vectors or arrays L and R, i.e. 1 2,3 0 4 is 1 2 3 0 4
Drop	L↓R	Removes the first (last) L items of array R, if L>0 (<0)
Outer product	L∘.f R	Applied f between all pairs of items from L and R
Compress	B/R	Selects subarrays along last axis according to boolean vector B
Reshape	LρR	Generates an array of shape L with the items of array R
Pi times	○R	The product of π and R
Circle functions	L○R	L specifies a trig. function on R, i.e. 1○R is sin R
Reduce	f/ R	For f=+, it is sum; f=×, it is product
Inner product	Lf.g R	For f=+, g=×, L +.× R is matrix multiplication
Magnitude	| R	Absolute value of R
Rotate	L⊖R, L⌽R	Rotate array R |L positions along the first or last axis
Conditional jump	→(cond)/L	Jump to the line with label L if the condition is true

References

1. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**(3), 137–147 (1976)
2. Asanoic, K., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
3. Blume, W., et al.: Parallel programming with polaris. IEEE Comput. **29**, 78–82 (1996)
4. Dave, C., Bae, H., Min, S., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multi-cores. IEEE Comput. 36–42 (2009)

5. Chen, H., Ching, W.-M., Zheng, D.: A comparison study on execution performance of MATLAB and APL. Vector J. Br. APL Assoc. (to appear)
6. Ching, W.-M.: Program analysis and code generation in an APL/370 compiler. IBM J. Res. Dev. **30**, 594–602 (1986)
7. Ching, W.-M., Xu, A.: A vector code back end of the APL370 compiler on IBM 3090 and some performance comparisons. In: Proceedings of APL'88 Conference, pp. 69–76 (1988)
8. Ching, W.-M., Ju, D.-c.: Execution of parallel APL on RP3. IBM J. Res. Dev. **35**(5/6), 767–777 (1991)
9. Ching, W.-M., Carini, P., Ju, D.: A primitive-based strategy for producing efficient code for very high level programs. Int. J. Comput. Lang. **23**(3), 41–50 (1993)
10. Ching, W.-M., Ju, D.: An APL-to-C compiler for the IBM RS/6000: compilation, performance and limitations. APL Quote Quad **23**(3), 15–21 (1993)
11. Ching, W.-M., Katz, A.: An experimental APL compiler for a distributed memory parallel machine. In: IEEE Proceedings of Supercomputing'94, Washington, D.C., pp. 59–68 (1994)
12. Guibas, L., Wyatt, D.: Compilation and delayed evaluation in APL. In: Proceedings of 5th ACM Symposium on Principles of Programming Languages, pp. 1–8 (1978)
13. Gupta, M., Midkiff, S., Schonberg, E., Seshadri, V., Shields, D., Wang, K., Ching, W., Ngo, T.: An HPF compiler for IBM SP2. In: Proceedings of Supercomputing 95
14. Iverson, K.: Turing award lecture: notation as a tool of thought. Commun. ACM **23**(8), 444–465 (1980)
15. Ju, D., Lin, C., Carini, P.: The classification, fusion and parallelization of array Language primitives. IEEE Trans. Parallel Distrib. Syst. **5**(10), 1113–1120 (1994)
16. Li, J., Ma, X., Yoginath, S., Kora, G., Samatova, N.: Transparent runtime parallelization of the R scripting language. J. Parallel Distrib. Comput. (to appear)
17. Liu, Y., Stoller, S., Li, N., Rothamel, T.: Optimizing aggregate array computations in loops. ACM Trans. Program. Lang. Syst. **27**(1), 91–125 (2005)
18. Novillo, D.: OpenMP and automatic parallelization in GCC. In: GCC Developers' Summit, Ottawa, Canada, June 2006
19. International Organization for Standardization: ISO Draft Standard APL. APL Quote Quad **14**(2), 7–272 (1983)
20. van der Pas, R.: An Overview of OpenMP 3.0. Sun Microsystems, Santa Clara, CA (2009)
21. The MathWorks: Parallel computing toolbox 4. User's guide (2010)