1992

Experiments with Program Unification on the CRAY Y-MP

Ling-Yu Chuang

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:
92-067

EXPERIMENTS WITH PROGRAM UNIFICATION ON THE CRAY Y-MP

Ling-Yu Chuang
Vernon Rego
Aditya Mathur

CSD-TR-92-067
September 23, 1992

Experiments with *Program Unification* on the CRAY Y-MP *

Ling-Yu Chuang, Vernon Rego and Aditya Mathur
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Program Unification is a technique for source-to-source transformation of code for enhanced execution performance on vector and SIMD architectures. This work focuses on simple examples of program unification and attempts to demonstrates that program unification has promise as a practical technique for improved program performance. Using simple examples to explain how unification can be done, we outline two experiments in the simulation domain, namely Monte Carlo and discrete event simulation that can benefit from unification. We conduct empirical tests of unified code on a Cray Y-MP multiprocessor to show that unification can improve execution performance by a factor of roughly 8 for the given application. The technique is general in that it can be applied to computation-intensive programs in various application domains.

KEYWORDS: program unification, parallel processing, vectorization, multitasking, simulation, Cray Y-MP.

*This research was supported by the National Science Foundation, under Grant No. ASC-9002225.

1 Introduction and Motivation

Vector and SIMD multiprocessor architectures have proven to be very effective frameworks for supporting computational problems in a variety of scientific endeavours. Coupled with advances in compiler technology and the decreasing cost of MIMD multiprocessors, vector and SIMD architectures are predicted to be an essential component of massively parallel architectures. Such architectures will consist of hundreds, or even thousands of interconnected processors, each supporting either vector or SIMD processing capability.

The experimental efforts described here are motivated by the potential for improving program performance on vector machines under certain conditions. While the techniques that we describe also apply to SIMD machines (with slight modification), the experiments reported here are oriented towards vector machines, in particular, the Cray Y-MP. Given that our ideas show potential for improved program performance on vector or SIMD machines, it follows that they have potential for improved program performance on large scale MIMD processing when such processing also supports vector or SIMD features.

The performance enhancement techniques that we experiment with in this work originated from a recognition of two facts. First, vector supercomputers such as the Cray Y-MP perform best on an application when the application is able to fully exploit the machine's vector pipes and functional units during execution. Second, compilers on such machines are able to extract parallelism from a given program only up to a point; if a compiler is unable to vectorize or concurrentize a loop, the user may be able to help the compiler by either providing more information to the compiler or rewriting code. However, the amount of parallelism present is dictated by the application's peculiarities and the compiler cannot extract any more parallelism than that which is present in the code.

In vector processing, the major emphasis tends to be the improvement of parallelization *within* a piece of code. A Cray Y-MP user typically evaluates the performance of his code by examining the extent to which the compiler vectorizes or concurrentizes the code; he evaluates the performance of the machine on his code by measuring the MFLOPS (Mega floating point operations per second) it attains during execution. Thus, if a user's code exhibits poor vectorizability (and a correspondingly

low MFLOPS, the user is led to believe that because of the two points mentioned above, little can be done to improve the performance of his code without the invention of a completely new algorithm (for his problem) and code that the compiler indicates has better vectorizability.

A Cray (or other vector machine) user often cannot avoid significant amounts of scalar computation and data dependencies in vector data structures. Consequently, his code will inhibit certain vector computations and preclude the efficient dispatchment of work across processors. Consequently, the search for an improved algorithm and corresponding code with improved vectorizability can be futile.

In those situations where code exhibits poor vectorizability, it is possible that a significantly improved performance can be had. However, this can happen only if the application possesses an important characteristic, namely, that the user requires this application to execute several times. A classic example of this situation is an application that is run sequentially on different data. The class of applications possessing this property includes discrete event and Monte Carlo simulation, where the term data implies parameters, random number seeds, or both.

The technique that we have proposed suggests that improved vectorizability and improved performance can result when the original piece code is transformed into a new piece of code such that a single execution of the transformed code on all data inputs simultaneously effectively substitutes for the execution of the original code on different data sequentially. This transformation technique is called *Program Unification* [10]. To demonstrate the effectiveness of program unification, we use it in two different applications and measure its performance. The technique speeds up one of these two applications by a factor of 8, on a single processor of the Cray Y-MP.

The remainder of the paper is organized as follows. The technique of program unification is briefly reviewed in Section 2, and the experiments and supporting results given in Section 3. We conclude briefly in Section 4.

2 Program Unification

Program unification is a scheme which exploits the parallelism that arises when multiple instances of a program are transformed into a single instance. The transformation satisfies the required

property that the output of the original program on all its data is also obtained by the transformed program on the same data [10]. The transformed program is thus made up of *components*, each of which is effectively an instance of the original program and its data.

If we label the original program as P, and the different data items as D_1 through D_N, then the transformed program $\mathbf{P}$ acts on the data set (D_1, ..., D_N) to obtain the same results as given by applying P to D_1, P to D_2, and so on up through P to D_N. For $\mathbf{P}$ to give the correct output, it must be the case that during the execution of $\mathbf{P}$ on its data, its components ($\mathbf{P}_j, D_j$), must traverse the same execution paths as the individual programs (P, D_j), respectively, for $1 \leq j \leq N$. In the rest of the paper, we will show that program unification can be efficient at exploiting the parallelism that arises when one is prepared to replace the sequential execution of P on data items D_1 through D_N by the execution of $\mathbf{P}$ on the same data items. In other words, improved performance can be had through use of $\mathbf{P}$.

2.1 A Simple Example

In Figure 1 can be seen a very simple program. All it does is to read in a mileage value, convert it into an equivalent value in kilometers, and print out the converted value. Note that the program does not lend itself to vectorization or multitasking. Suppose that we want to convert a certain number, say 10, of mileage values into the same number of equivalent values in kilometers. The natural procedure entails 10 repeated executions of this program. A convenient scheme for achieving such repeated executions in the Unix environment is through the use of a shell script (see Figure 2). However, doing this on a Cray Y-MP or any other vector multiprocessor will not yield a program which can be vectorized or multitasked; through use of the shell script, it will still execute sequentially.

An alternative scheme and one that is often confused with the program unification technique is one that utilizes a loop. This one do-loop scheme places a loop around the body of the program, as shown in Figure 3. This technique is often mistaken for program unification because it is assumed that unification improves vectorization through a single simple and explicit loop-level enhancement to the original program which is $\mathbf{P}$. Indeed while unification improves vectorization through loop-level enhancements, these enhancements exhibit quite a wide range in complexity and

```
      program  convertseq
C Declaration
      real km, mile
C Input
      read *, mile
C Computation
      km = 1.64*mile
C Output
      print *, km
      end
```

Figure 1: The non-unified program "convertseq.f"

```
#!/bin/sh
for i in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10; do
  convertseq < data$i >> outfile
done;
```

Figure 2: The shell script

```
      program  convertloop
C Declaration
      real km, mile
C Input
      do 10 i = 1, 10
        read *, mile
C Computation
        km = 1.64*mile
C Output
        print *, km
 10   continue
      end
```

Figure 3: The one do-loop scheme "convertloop.f"

```
      program  convertmul
C Declaration
      real km(10), mile(10)
C Input
      do 1 i = 1, 10
        read *, mile(i)
 1    continue
C Computation
      do 10 i = 1, 10
        km(i) = 1.64*mile(i)
 10   continue
C Output
      do 2 i = 1, 10
        print *, km(i)
 2    continue
      end
```

Figure 4: The unified program "convertmul.f"

are often implicit. Thus, this one do-loop technique gives a program whose output is equivalent to **P** but whose execution is sequential; the I/O statements within the loop preclude any possibility of vectorization, and the scalar data (variables `km` and `mile`) in the program inhibit vectorization. Hence use of the loop does not increase the compiler's ability to vectorize the code. Although the example is very simple minded, it suffices to demonstrate that using a loop in this way cannot overcome vectorization inhibitors such as non-vectorizable statements and scalar data.

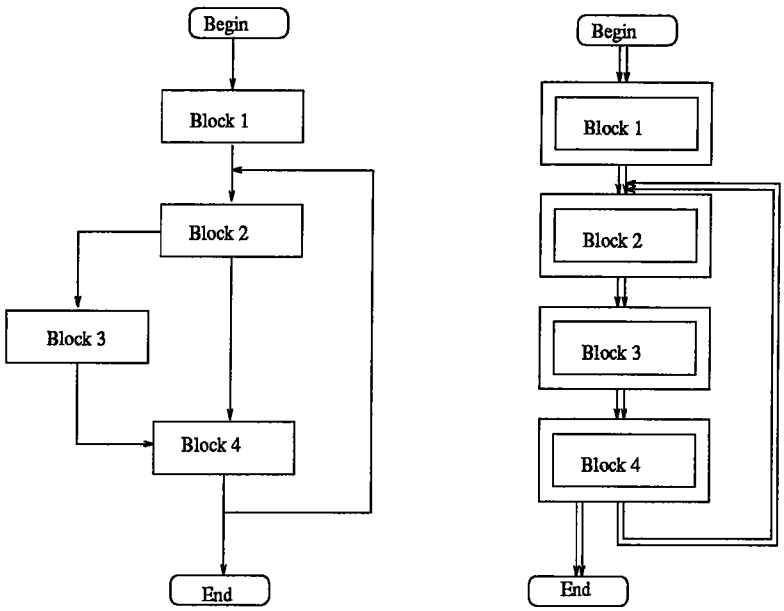
The program unification technique increases parallelism by merging the independently executing program instances (where each instance is program P acting on a data item) into a coherent whole; parallelism exploitation is done across the the corresponding components of the resulting program **P**, where each component is logically a program instance. The process of unifying the independent components into a single coherent program **P** has suggested the term "unification". In Figure 4 is shown a unified version of the previous example. The second loop in the program is now vectorizable, and as a result the parallelism in the program increases. In this example, the unified program contains 10 independent components.

To unify a program, two major source-to-source transformations are needed. First, we must replace all scalar data items by equivalent vector items and structures so that these vectorize. For example, the variable `km` in Figure 1 is replaced by `km(10)` in Figure 4. Second, we break a program into blocks so the program now consists of units which may vectorize independently. Loosely speaking, a block is a segment of code between statements responsible for branches in control flow, and contains no nonvectorizable constructs. A formal definition can be found in [10]. Segments that contain I/O statements or other nonvectorizable constructs should be put into separate blocks. As shown in Figure 4, portions of the code responsible for computation such as the loop labeled 10, can now be vectorized and multitasked. This is possible because the I/O statements are put in different loops. However, increasing the number of such blocks in a program introduces extra loop setup overhead. Therefore, blocks should be large enough to vectorize well and keep increased loop setup overhead low.

2.2 Partition Vector and Flow Control

Since the components of a unified program execute along independent trajectories (taking data-dependent paths), their execution paths may be different in general. In Figure 5.a is shown a control flow graph for a sequential program where execution starts from Block 1. After Block 1, control moves to Block 2. At this point, depending on the input data, some program components execute Block 3 before moving to Block 4, while others skip Block 3 and move directly to execute Block 4 after executing Block 2. As in the case of Block 2, Block 4 also offers the possibility of path divergence. While some program components move back to Block 2 after Block 4, others may terminate their executions at Block 4.

As shown in the previous example, unified programs may exhibit execution paths that are very different from one another. To handle this, unified programs utilize a *partition vector.* A *partition vector* is a vector whose elements contain some information relevant to the control of unified program flow. Here, we take a partition vector to be a boolean vector with as many elements as components in the unified program. Each element in the vector is a logical value representing the status of a component. A "true" value indicates that the corresponding component is active and a "false"



a. the sequential program b. the unified program

Figure 5: A control flow graph for both the sequential and the unified programs

indicates the contrary. The partition vector is *set* at each branch statement and is checked at the entrance to each block. When unified program flow control reaches such a branch point, only active program components are allowed to enter the following block, while inactive components simply wait for the active ones to finish execution of this block.

In Figure 5.b is shown the control flow graph of the unified program for the above example. Note that execution of a block here is essentially the vector execution of all components of the unified program. Initially, all components are assigned the value "true" in the partition vector. Program control starts at Block 1 and then moves to Block 2. Since there is a possible program path divergence at Block 2 in the original program, some components of the unified program may need to execute Block 3 while others may not need to execute Block 3. Though Figure 5.b may seem to indicate that Block 3 is scheduled to be executed by all components after Block 2, the presence of the *partition vector* allows certain components to avoid execution of Block 3. This is done by setting the partition vector at the end of Block 2. We assign the value true to those

components which need to execute Block 3 and the value false to the others. Since the partition vector is examined prior to execution of Block 3, only those components which have partition vector elements set to true (and are correspondingly *active*) are allowed to execute the code in Block 3. After execution of Block 3, the partition vector is reset and program control moves on to Block 4. After execution of Block 4, a component which terminates its execution is permanently assigned a value of "false" in the partition vector and becomes inactive for the rest of the unified program's execution. Execution of the unified program continues as long as at least one of the program components still remains active. Since the time that inactive components spend waiting may degrade the performance of a unified program, high program path divergence may lead to less efficient unified programs. Correspondingly, low program path divergence leads to very efficient unified programs.

2.3 Applicability of Program Unification

There are two strategies for the use of *program unification* in compute intensive applications. One strategy is the **individual** strategy, where an application is required to execute on several different data sets; the individual results obtained by executing the application on each data set is of importance. The unified program takes in all the data sets simultaneously as a single input and produces the results corresponding to all the different data sets upon its termination. In this way a single execution of the unified program replaces multiple independent executions of the original program. An example of this arises in any application that requires parametric input and yields a result that is a function of the input.

The other strategy is the **aggregation** strategy where the result obtained by pooling the results corresponding to the independent data sets is of importance. An example of this arises in the simulation of stochastic processes, where each simulation generates a different trajectory from the same stochastic process through use of different random number seeds. Though simulation programs may take different trajectories, they execute the same code, and some function of their aggregate results (i.e., for example, an average) is of more importance than particular trajectories. Thus, instead of executing a single simulation program over a long time to repeatedly generate

independent trajectories, we execute a unified program which generates several independent trajectories simultaneously. Any required aggregation can be performed during or at the end of unified program execution.

3 Experiments

One acceptable definition of a Monte Carlo simulation is a simulation in which it is necessary to generate at least one stream of random observations from some specified probability distribution [7]. Monte Carlo simulation is widely used for solving a variety of computationally difficult or analytically intractable problems; an example of such a problem in statistics arises in estimating the critical values or the power of a new hypothesis test [8]. As another example, most queueing and telephone traffic simulations are of the Monte Carlo type [7]. It is well-known that Monte Carlo programs perform poorly on vector machines in the absence of special handling [12]. Moreover, simulation programs of the Monte Carlo type need to be repeatly executed for reasons of statistical accuracy, sensitivity studies, model verification, etc. Therefore, Monte Carlo codes fit very well within the range of applicability of unification, and as a result stand to benefit from improved vectorizability.

In this section we present the results obtained by using the program unification technique of two different simulation applications. These experiments were conducted on the Cray Y-MP at the University of Illinois Supercomputing Centre. All programs were written in Fortran and preprocessed by the Cray Y-MP autotasking preprocessors, **fpp** and **fmp**. **Fpp** analyzes Fortran code and inserts directives for autotasking and vectorization. **Fmp** translates autotasking and microtasking directives into multitasking library routines. [5] After preprocessing, the programs are compiled using **cf77** with a **cft77** option "-dp", which disables double precision computation on the Cray Y-MP [4]. In order to maximize the potential for vectorization, all function and subroutine calls are expanded at compile time using the on-line expansion facility on the Cray Y-MP [5].

Through our experiments, we show that a unified program exhibits superior execution performance compared to executing the program instances corresponding to the components of the unified

program separately. The one do-loop approach (see Section 2), which executes each component sequentially is used as a basis of comparison, since this is the technique a typical vector machine user would resort to in the absence of a program unification tool. Speedup is the criterion of merit and is defined as follows:

Let N = number of components

T_1 = CPU time spent on N sequential programs in one-do-loop approach

T_n = CPU time spent on the execution of the unified program with N components.

$S_p = T_1/T_n$

To observe the relationship between speedup and the number of program components, we conducted experiments for unified programs with a varying number of components. Since the vector length of the Cray Y-MP is 64, we restrict our sizes to be factors and multiples of 64. The job accounting facility [3], `ja`, is used to obtain the CPU time from the system.

3.1 Multi-dimensional Integral Estimation

Computing multi-dimensional integrals generally requires a substantial amount of computational effort. Monte Carlo methods are known to be more efficient than analytical techniques in estimating multi-dimensional integrals when the number of dimensions is high [6, 9]. One such Monte Carlo method is the *sample mean* method. Suppose that we want to estimate the integral of a function $h(\mathbf{x})$, assumed to be bounded and non-negative in a domain $\mathbf{R}$. We first define a density function $f(\mathbf{x})$ in the same domain. The estimation of the integral of $h(\mathbf{x})$, $\hat{\mathbf{I}}$, can then be obtained from Equation (1) [11, 6],

$$\hat{\mathbf{I}} = \frac{1}{n} * \sum_{i=1}^{n} \left(\frac{h(\mathbf{x}_i)}{f(\mathbf{x}_i)}\right) \tag{1}$$

and $\mathbf{x}_i, i = 1, n$ are vectors randomly chosen from the domain $\mathbf{R}$.

In Figure 6 is shown the algorithm for the implementation of the sample mean method. In the algorithm, sampling (computation) continues until a specified statistical accuracy is obtained. Consequently, the execution time of the program is not necessarily fixed, with different random number seeds yielding program paths with different execution times. Since the code exhibits no possibility of program path divergence during execution, the duration of program execution is the

BEGIN
n = 0
REPEAT
n = n + 1
Sample **x**
$S_n = \frac{h(\mathbf{x})}{f(\mathbf{x})}$
Compute the Mean of $S_i, i = 1, n$
Compute the Variance of $S_i, i = 1, n$
UNTIL (the confidence interval of a desired precision is reached)

END

Figure 6: Algorithm for multi-dimensional integral estimation

only factor that can cause one program component's behaviour to differ from another.

In performing our experiments, we apply both the **individual** and the **aggregation** unification strategies to the Monte Carlo integration code. Program termination in the non unified and unified case coincides with the computation of a confidence interval with a specified precision for the estimate. In the **individual** strategy, program components work independently on their data, taking different amounts of time to report final results (i.e., an estimate and a confidence interval) for their data. Due to these different run times, program divergence results.

In contrast, the **aggregation** strategy forces components to pool results during unified program execution so that all components terminate on the same condition. Because of this, there is no program path divergence.

Our measurements indicate that while the original (non unified) program exhibits zero vectorizability (i.e., the compiler is unable to vectorize it), the **individual** strategy exhibits 71% vectorizability, and the **aggregation** strategy exhibits 56% vectorizability under compilation. It is important to recognize that higher vectorizability does not imply higher performance because performance also depends on the amount of program path divergence arising during unified program execution. The latter strategy exhibits lower vectorization due to the presence of extra code required for result aggregation. Despite this, the **aggregation** scheme usually exhibits better execution performance than the **individual** scheme because it possesses no path divergence.

The results of our experiments are presented in Figures 7,8,9, and 10. In Figures 7 and 8 are

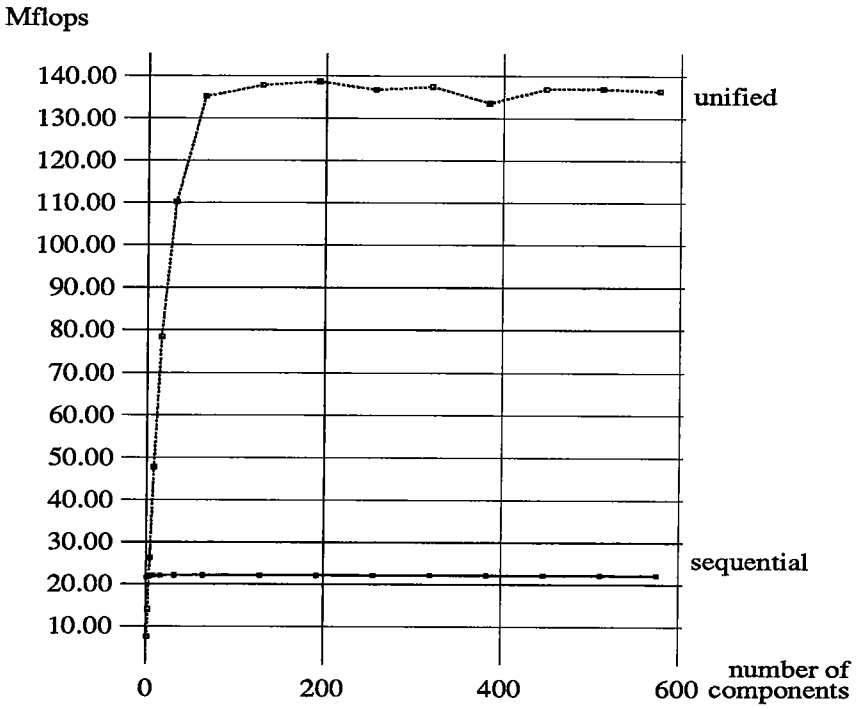


Figure 7: Mflops for multi-dimension integral estimation (**individual**)

shown the Mflops for a single Cray Y-MP processor under the **individual** and the **aggregation** schemes, respectively. In Figure 9 is shown the speedup obtained for both strategies, assuming use of only a single processor. In Figure 10 is shown the speedup obtained for the **aggregation** scheme under use of one processor and and four processors. Thus, the unified **aggregate** scheme exhibits a higher Mflops than the **individual** scheme. The higher Mflops and zero program path divergence are responsible for the superior performance exhibited by the **aggregation** scheme.

3.2 Single-Sever Queueing System

A single server queueing system [7] consists of a single server and a potentially unlimited number of customers. Customers arrive randomly and queue for service. The server chooses to serve customers in some predetermined order, for example first-in first-out. Customers arriving to find the server busy simply wait (for example, in a first-in first-out queue) to obtain service, leaving the system only after they are served. Service durations of these customers may be random. Since the underlying stochastic process may be conveniently described by focusing on the system at the discrete instants of customer arrival and end of service, simulations of such processes based on

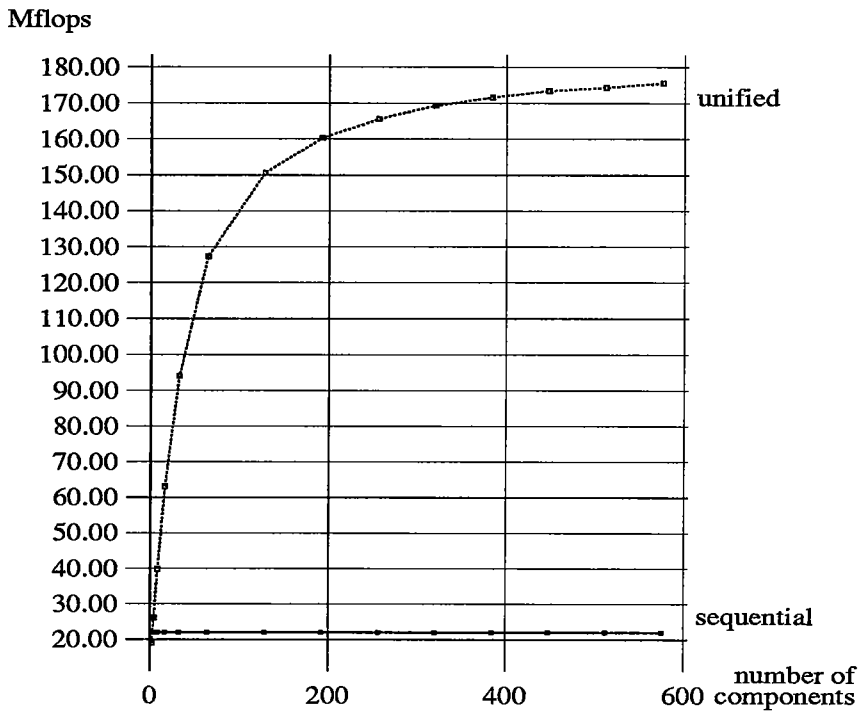


Figure 8: Mflops for multi-dimension integral estimation (**aggregation**)

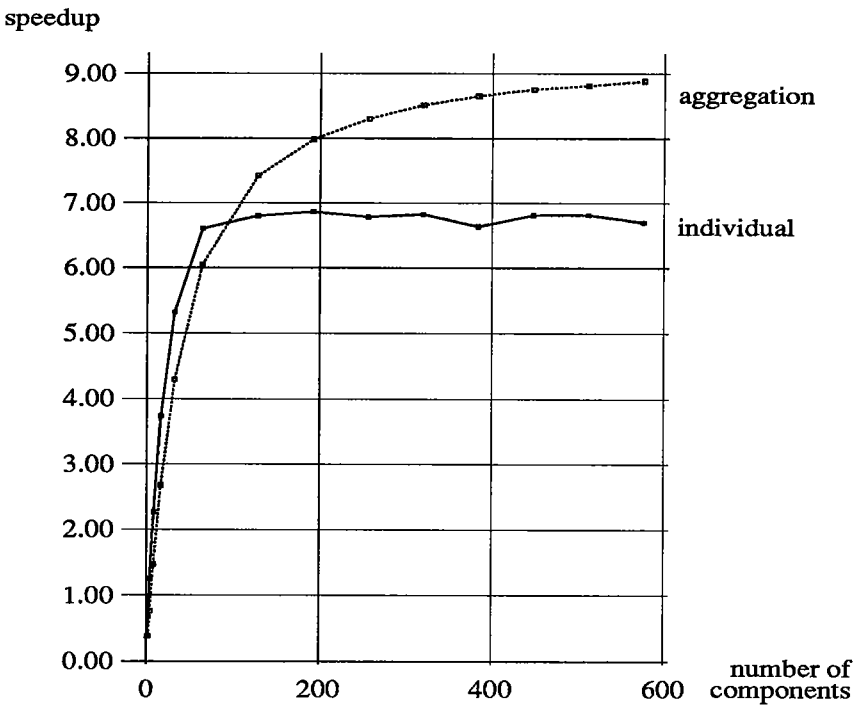


Figure 9: Speedup for multi-dimension integral estimation (one processor)

such discrete events are called discrete event simulations. The events of customer arrival and end

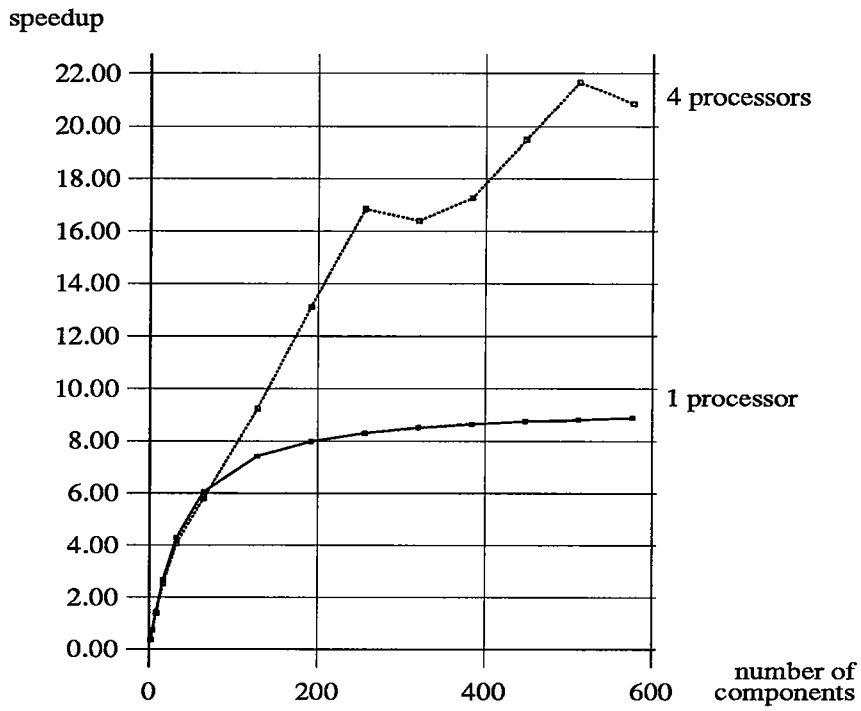


Figure 10: Speedup for multi-dimension integral estimation (**individual**)

of service are convenient descriptors [8] for times at which the system state (e.g., the number of customers in the system) changes.

There are two basic events in the single-server queueing system, namely, the *arrival* event (**A** event) and the *end-of-service* event (**E** event). For proper event handling during simulation processing, a priority queue [1] is used to maintain pending events. We use the heap [1] to implement a priority queue because it is known to be an efficient structure for simulation event processing. More importantly, we have found it to exhibit excellent performance when unified and executed in a general setting [2].

A sequential program for the the single-server queueing system simulation begins with simulation initialization and generation of the first customer arrival. The program then repeatly executes routines in the following sequence: get a next event from the event list, process the event according to its type, and produce statistical data (and terminate upon a specified condition, such as a confidence interval with certain precision). In Figure 11.a is shown the control flow of the sequential program. Here, **I**, **G**, **A**, **E**, and **S** indicate "Initialization", "Get next event processing", "Arrival event processing", "End-of-service event processing", and "Statistical computation", respectively.

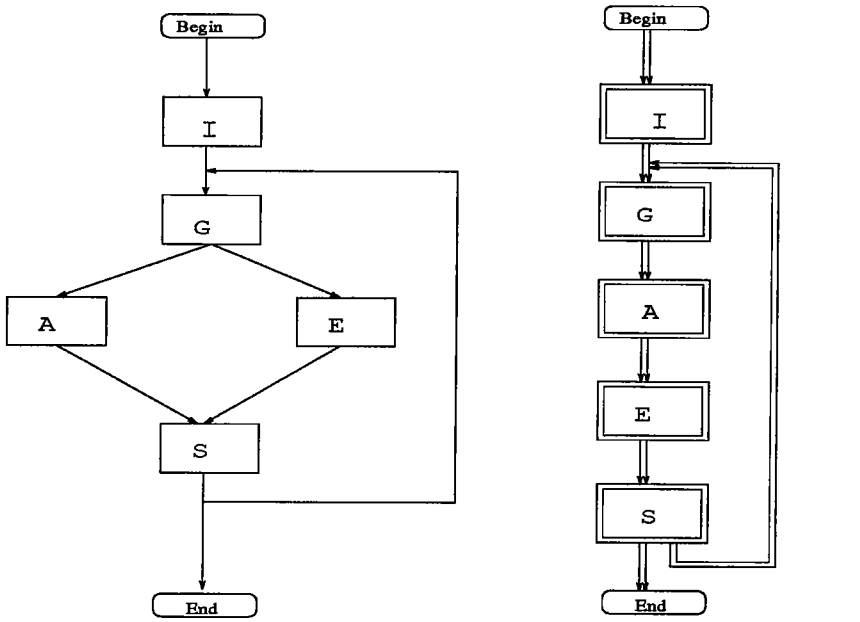


Figure 11: The control flow graph for the single-server queueing system

At **G**, the program retrieves from the event list the next event, which has the earliest time of occurrence. If the event retrieved is an arrival event, program control moves to **A**. Otherwise program control moves to **E** for an end-of-service event. New arrival events and of end-of-service events are continually generated by blocks **A** and **E**. After processing an event, program control moves to **S**, where statistical data is computed. If the statistical computation meets some specified accuracy, the simulation terminates; otherwise, control moves back to **E** and the same process is repeated.

In Figure 11.b is shown the unified program for the single-server queueing system. All components virtually execute every block concurrently. As stated in the previous section, if a component is not supposed to execute a certain block, it is simply set inactive in the partition vector and waits while the unified program executes that block. Therefore, after execution of **G**, components determining their next event to be an end-of-service event remain inactive while unified program control is at **A**; they become active when program control moves to **E**. In like fashion, those components which determine their next event to be an arrival event after execution of **G** must be active and

execute in **A**, but become inactive in **E**.

We performed two experiments with different program termination conditions for the single server simulation application. In one of these, denoted by *SSQa*, the simulation terminates when an estimate of customer mean waiting time is obtained with a confidence interval of a specified precision. This means that if different program components are made to work on different data (say for example, different arrival rates or service rates or both), then they will not terminate at the same time, and consequently will exhibit execution length path divergence. This is in addition to program path divergence caused by program components taking different execution paths upon leaving **G**.

In the other experiment, denoted by *SSQb*, the simulation terminates when a certain number of customers has been processed by every unified program component. Since each component processes the same number of customers, each component terminates execution at the same time. Since there is execution length path divergence in the former scheme, it is reasonable to expect that if both schemes vectorize to roughly the same extent, the latter scheme will exhibit superior performance. Our measurements indicate that 85% of the unified code for *SSQa* vectorized under compilation, while 84% of the unified code for *SSQb* vectorized under compilation. Only 1% of the program vectorized under the one-do-loop scheme.

The experimental data for this set of experiments is presented in Figures 12,13, and 14. In Figure 12 and Figure 13 are shown the single processor Cray Y-MP Mflops for *SSQa* and *SSQb* unified program executions, respectively. In Figure 14 is shown the speedup obtained for both experiments when one processor and four processors are used, respectively. As expected, it can be seen that *SSQb* exhibits superior performance.

4 Conclusion and Future Work

Through experiments we have demonstrated that program unification can be used to improve the performance of programs on vector machines. In one case, the performance improved by a factor of roughly 800. In combination with previous work [9, 2, 11], the results of this study suggest that program unification is a viable and practical technique for enhancing program performance on

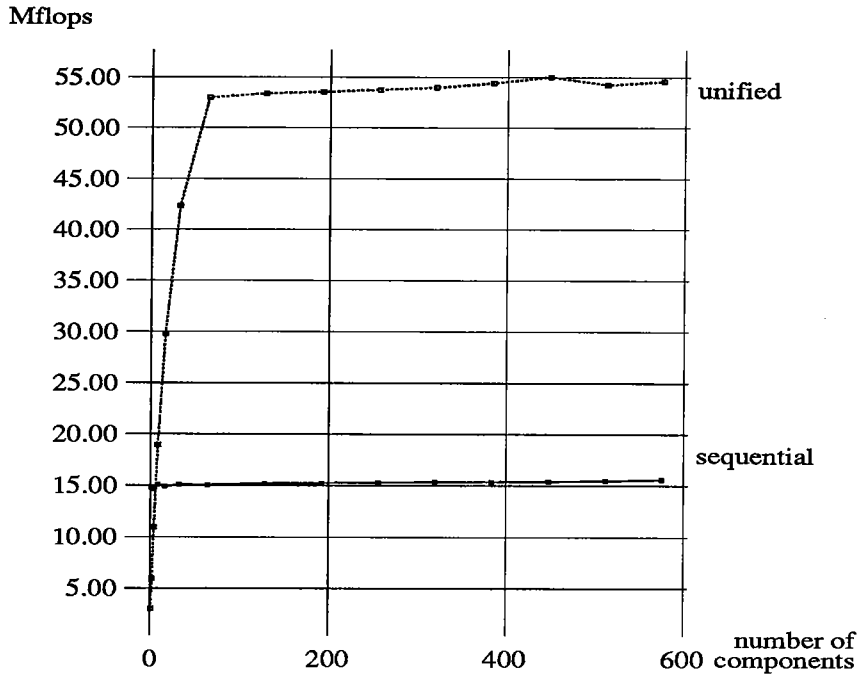


Figure 12: Mflops for single server queueing system ($SSQa$)

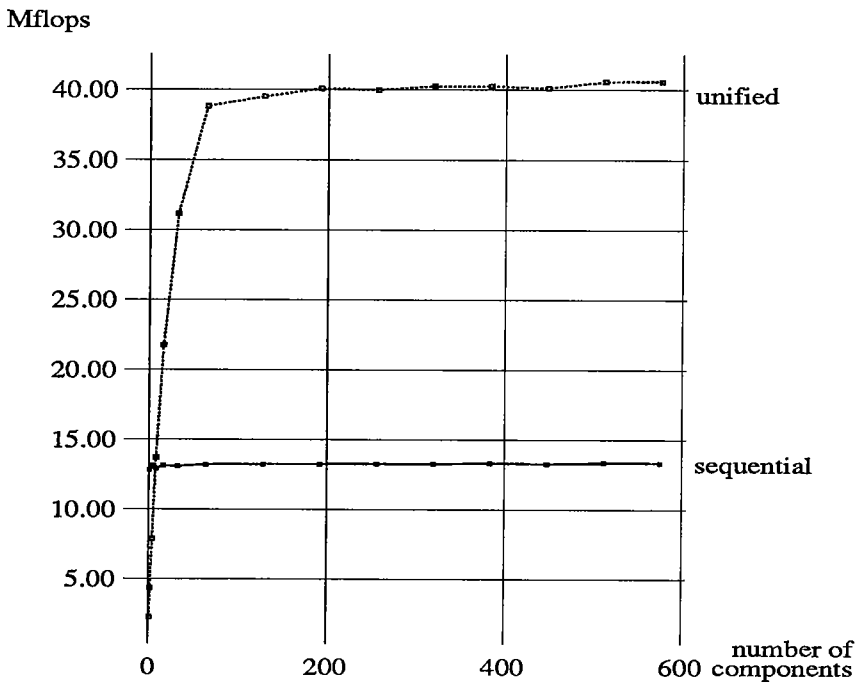


Figure 13: Mflops for single server queueing system ($SSQb$)

vector machines. Our current work involves more detailed empirical studies of program unification

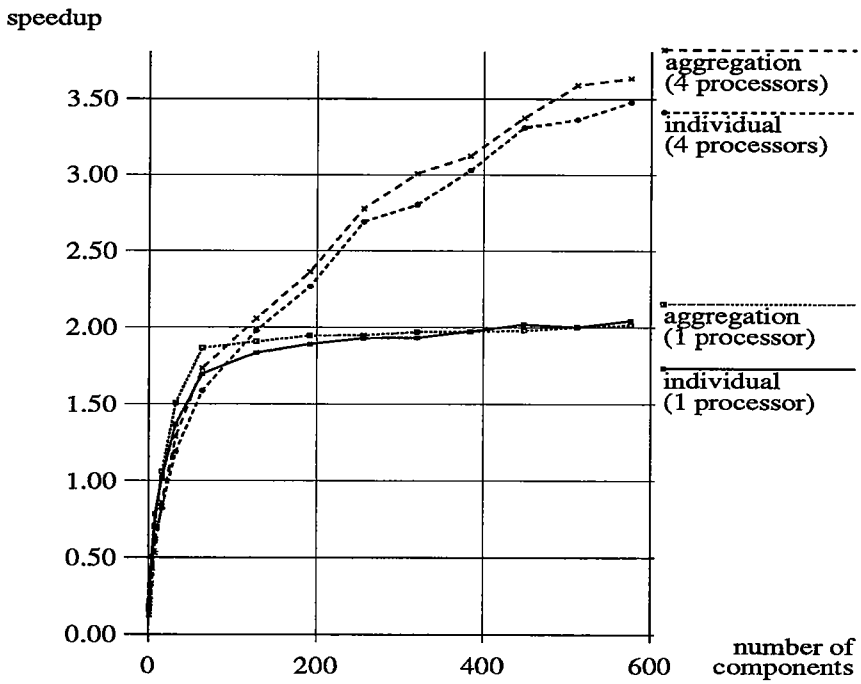


Figure 14: Speedup for single server queueing system

and related scheduling issues on vector machines and SIMD machines, and a program unification tool for both architectures.

References

[1] A. V. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass, 1974.

[2] L. Chuang, V. J. Rego, and A. P. Mathur. An application of program unification to priority queue vectorization. Technical Report CSD-TR-92-034, Computer Sciences Department, Purdue University, 1992.

[3] Cray Research, Inc. *UNICOS Performance Utilities Reference Manual (SR-2040B)*, 1989.

[4] Cray Research, Inc. *CF77 Vol1: Fortran Reference Manual (SR-3071)*, 1990.

[5] Cray Research, Inc. *CF77 Vol4: Parallel Processing Guide SG-3074*, 1990.

[6] D.S. Fok and D. Crevier. Volume estimation by Monte Carlo methods. *Journal of Statistical Computation and Simulation*, Vol. 31:223–235, 1989.

[7] D. Gross and C.M. Harris. *Fundamentals of Queueing Theory.* John Willey and Sons, New York, NY,, 2nd edition, 1985.

[8] A.M. Law and W.D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 1982.

[9] V. Rego, L. Chuang, and A. Mathur. Unified stochastic simulations for vector machines: Empirical results. Technical Report CSD-TR-92-030, Computer Sciences Department, Purdue University, 1992.

[10] V. Rego and A. P. Mathur. Exploiting parallelism across program execution: a unification technique and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, October 1990.

[11] V. J. Rego, L. Chuang, and A. Mathur. Concurrent stochastic simulations: Experiments with unification. In *Fifth Annual Canadian Supercomputing Symposium*, Fredericton, N. B., Canada, 1991.

[12] R. Sarno, V. C. Bhavar, and P. Banerjee. Design and analysis of vectorized Monte Carlo codes. Technical Report CSTR 89-048, University of New Brunswick, Canada E3B 5A3, 1989.