A SOFTWARE TOOL FOR THE AUTOMATIC GENERATION OF MEMORY TRACES FOR SHARED MEMORY MULTIPROCESSOR SYSTEMS.

Saurabh Gupta
Cap Gemini America,
25 Commerce Drive,
Cranford, NJ 07016

and

Rami Melhem
Department of Computer Science,
University of Pittsburgh,
Pittsburgh, PA 15260

Abstract

We have developed a software tool which allows for the simulation of parallel programs on shared memory, multiple instruction multiple data (MIMD) machines in the absence of such a machine. Our aim is to obtain a history of requests generated by parallel programs to the shared memory. Specificly, the result of the simulation is a memory trace which records the history of memory access. The trace contains information such as the time of the request, the identification of the processor making the request, and the specific memory location accessed. The software tool consists of a translator-augmenter which translates and enhances programs written in a parallel version of C into standard C programs which we call *trace generators*. When compiled and executed as C programs, the trace generators generate static memory traces. These traces can be used to simulate and analyze memory contentions in different memory architectures and to evaluate and compare different parallel algorithms.

Introduction

One of the important factors to be taken into consideration in the design of a multiprocessor system is the interconnection mechanism that will be used to provide interaction between the various processors and memory modules. The interconnection network is a critical factor in the matching of memory bandwidth and processing power and has often been the limiting factor (Patel 1981) in the speedup provided by such systems. The most common types of switching devices used in commercial multiprocessor systems thus far have been the Time Shared Bus, the Crossbar Switch and Packet Switched Networks. The Time shared Bus (Bain and Ahuja 1981) has been used on several commercial multiprocessor systems such as the Encore Multimax and the Sequent 8000 Balance. Crossbar Switches have been implemented on the CMU Cmmp system (Wulf and Bell 1972). Switching networks have been quite successful in this respect and a number of commercial machines such as the NYU Ultracomputer (Gottlieb et al.1983) which is based on the Omega Network, BBN Butterfly and the Denelcor HEP (Smith 1978) are based on them. A different approach would be to use a memory bus which has a very

high bandwidth. An optical bus may provide such a high memory bandwidth (Chiarulli et al. 1987) and (Kuck et al. 1977) thus alleviating the problem of memory contention.

In this paper we describe a software tool which enables the simulation of parallel programs running on shared memory multi-processor systems. The result of the simulation is a set of memory traces that describe the history of memory requests. Such traces have often been used to analyze different interconnection schemes and memory organizations (Lee et al. 1987). The tool described here allows the user to simulate a parallel program running on a multiprocessor system and generates traces for these programs without the need to actually run the programs on an actual parallel machine. The system is designed to be easy to use and its aim is to generate the actual memory traces with minimal effort on the part of the programmer. It requires the user to write parallel programs in a simplified subset of the 'C' language which is augmented with a parallel-do construct to express parallelism. We call this language *CPSIM*, an acronym for *C for Parallel Simulation*. The software tool consists of a translator-augmenter which translates and enhances programs written in *CPSIM* to a C program (trace generator) which is augmented with additional code and variables which keep track of factors such as the currently executing processor, elapsed time and shared memory requests. When compiled and executed as a normal C program the trace generator generates static memory traces for the parallel program. Figure 1 graphically illustrates the process of using the tool to create a trace generator and obtain a trace.

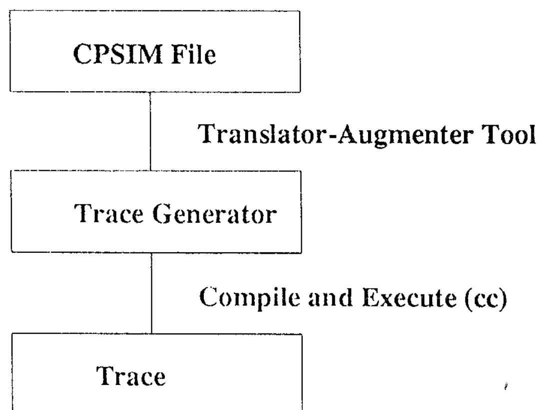

Figure 1 - The structure of the simulation tool

The parallel-do construct (*pardo* in *CPSIM*) enables a given set of processors to execute similar code. It also allows the processors to synchronize with each other as it logically defines the portion of code which have to execute by all processors before they can move on to a different part of the computation. Variables may be specified to reside either in local memory or in global (shared) memory. Specificly, a variable is stored in global memory if it is prefixed by a g_, otherwise it is stored in local memory.

Clearly, a trace resulting from the simulation incorporates any data-dependent delays. However, because it is obtained without explicit assumptions about the shared memory architecture, it cannot incorporate run-time delays that are caused by memory contention. These delays affect the execution of the parallel program and thus need to be

added to the trace. Given that they are only known when memory conflict is resolved, the task of inserting these delays is left to the memory analyzer. Many features are introduced, however, to simplify the insertion of these delays and to ensure the correctness of the trace.

A similar difficulty is caused by the synchronization primitives. Specifically, at a synchronization point, all processors have to wait for the slowest processor. Again, the synchronization delay depends on both the input data and the memory conflict. In order to solve this problem, the trace generator inserts breakpoints whenever such delay has to be inserted. At these points, the memory analyzer should align the local clocks of all processors, and shift the time component of future memory requests accordingly.

We will describe an implementation of the translator that was developed using the UNIX† tools Lex (A Lexical analyzer) and Yacc (A LALR parser). We will also present our solutions to the many technical problems that had to be resolved in order to obtain correct trace generators. The usefulness of the simulation tool will be demonstrated by an example for parallel matrix multiplication.

Model of Computation

The model of Computation is shown in the figure below. Each Processor in this system has a local memory and all processors are connected to a global shared memory via an interconnection mechanism. Processors can access any location in the global shared memory. However, they cannot access the local memory of another processor. All processors are identical and execute at the same frequency i.e. they have the same clock cycle and therefore any instruction when executed would require exactly the same cycle time to execute on any of the processors. Each processor is controlled by a global clock. However, the processors operate independently and have to be synchronized with each other explicitly in order to preserve dependencies in the program.

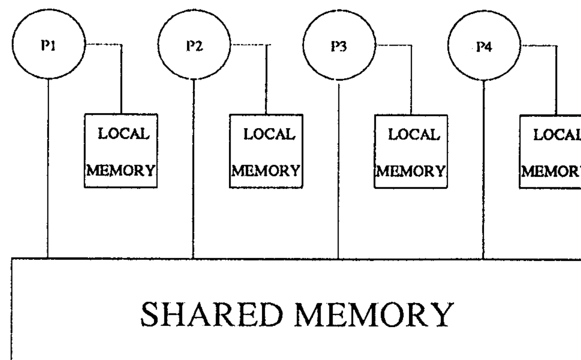

Figure 2 - The model of Computation

In order to simulate and determine the effectiveness of an addressing scheme we need to generate actual memory traces for some commonly used parallel algorithms. For each shared memory request made, the trace contains information such as the time of

† UNIX is a trademark of Bell Laboratories.

the request (in terms of machine cycles), the identification of the Processor making the request, whether the request is for Reading from memory or writing to memory and the specific memory location requested. Traces are generated only for shared memory requests.

As is the case with any statically obtained trace, the traces generated by our tool do not incorporate the many delays that may influence the execution of parallel programs on parallel machines. Such delays are usually caused by memory contention, processor synchronization or by I/O. These delays need to be added to the traces and are only known when the traces are used to analyze memory conflicts (caused both by memory organization and by interconnection structure). Hence, the task of inserting these delays is left to the memory analyzer. Many features are introduced to simplify the insertion of these delays and to ensure the correctness of the trace as described in later sections. An illustration of this problem is given below. Consider the following section of a memory trace:

Processor	Cycle	MemLoc
1	1200	345
2	1200	346
3	1200	356
4	1200	367
1	1205	400
2	1206	346
3	1205	357
4	1205	369

Let us assume that memory locations 345, 346, 400 and 367 belong to the same memory bank, say bank 1, and that locations 356, 357 and 369 belong to another memory bank, say bank 2. From the section of the traces above we can see that there is no guarantee that processor #2 will be able to access memory location 346 in the same cycle as processor #1 which is trying to access location 345 as they belong to the same memory bank. Memory contention may cause one of the processors (1 or 2) to wait while the other obtains access to the memory location. For example if processor 2 is allowed access to memory bank 1 and holds this bank for 2 machine cycles then processor 1 will be delayed by 2 cycles before it can access location 345. This delay would have to be added to the trace of processor 1 and it would then access memory location 400 at time 1207 and not 1205. It is an easy task for the memory analyzer to keep track of the delays

Proc	Cycle	Delay	Actual Cycle	Memloc
1	1200	0	1200	345
2	1200	0	1200	346
3	1200	0	1200	356
4	1200	0	1200	367
1	1205	2	1207	400
2	1206	0	1206	346
3	1205	3	1209	357
4	1205	1	1206	369

and insert them at the appropriate locations within the trace and hence shift the trace. The actual machine cycles for the above trace would be the sum of static times plus the delay caused and this scenario is depicted below. Another problem is caused by synchronization and this will be discussed later in this paper.

Source Language

CPSIM, a subset of 'C' is used as the base language in which the parallel programs are written. *CPSIM* includes most of the basic constructs in C such as the while and for loops, if statements, arrays, functions, arithmetic expressions, integer and character data types and preprocessor statements. Certain modifications have been made to the language to enable the programmer to write parallel programs for this model. The various features of *CPSIM*i are:

1) The ability to define variables to reside either in global (shared) memory or local memory. This is implemented by allowing only those variables which are prefixed by a g_ to be in global shared memory. All other variables which do not have a similar prefix are assumed to be local to the particular processor and it is the responsibility of the programmer to make this distinction. Variables stored in local memory are in effect copies of a single variable stored in the local memories of various processors and can contain different data pertaining to each processor. Data pertaining to variables stored in shared memory such as size, location and dimension are stored in a symbol table which is constructed and maintained by our software tool.

2) A parallel looping construct *pardo* has been added to express parallelism and enable a given set of N processors to execute similar code. The loop also allows the processors to synchronize with each other as it logically defines the portion of code which would have to be executed by all processors before they can move on to a different part of the computation. The *pardo* loop is similar to the for loop and has to be used for any portion of the program for which traces need to be generated. If the program executes on a parallel machine, then the completion of this loop automatically signals a synchronization to be performed among all processors (i.e. All processors wait for the slowest processor to complete the task before continuing).

3) The three statements *for, while* and *if* must continue as compound statements. This is required in order that an extra pair of braces need not be added by the translator for each statement and only simplifies the implementation.

4) During function calls the types of the formal and actual parameters must match. i.e. both the formal and actual parameters must be stored in the same type of memory (local or shared). For example, if the actual parameter is a reference to a shared memory location then the formal parameter must also be defined as a shared memory variable. Similarly if the parameter is a "call by value" then the formal parameter should be defined as a variable in the local memory of the processor.

5) The language does not recognize data types other than integer and character. New data types may be added by the modification of the grammar and lexical analyzer.

6) Output is not permitted by the language as the aim of our trace generator is to generate purely static traces and to avoid unpredictable I/O delays. *CPSIM* programs must be tested for syntactic correctness before they are submitted to the translator. This is easily done by replacing all *pardo* loops by *for* statements, adding appropriate statements to perform input-output and running the program through *cc*, the standard C Compiler.

7) The language reserves the use of the variables P and PROC. The variable P should be used by the programmer to define the number of processors. If the variable P is not defined by the programmer the translator assigns it a default value of 1 implying a uniprocessor system. PROC is used as a global variable which keeps track of the currently executing processor. The processor number can be determined within a function by looking up this global variable. The PROC variable should not be defined by the programmer. It is defined by the translator based on the number of processors P defined by the programmer. The translator is explained in some detail later in this paper.

An sample program written in this language is shown below. It is a trivial task in which 5 processors interact to calculate the sum of 10000 data items stored in shared memory. The *pardo* loop shows the region of code which is executed simultaneously by all processors. Traces will be generated for global memory accesses within *pardo* statements only. Note that the variables stored in global memory are prefixed by **g_**.

```
/* DEMO: Sum N items using P Processors */

#define N  10000              /* N items */
#define P  5                  /* P Processors */
#define elem  2000            /* items per processor */

main( )
{
      int g_data[N];          /* global data array */
      int g_sum[P];           /* global memory of each processor */
      int i, j;               /* local variables */
      int g_total;            /* total stored globally */

      /* each processor calculates its sum */
      pardo( i = 0; i < P; i++) {
            g_sum[i] = 0;        /* initialize sum */
            for(j = i*elem; j < i*elem + elem; j++) {
                  g_sum[i] += g_data[j];
            }
      }

      /* all processors should reach this point before proceeding */

      /* Processor 0 is used to calculate total sum */
      pardo (i= 0; i< 1; i++) {
            g_total = 0;
            for (j = 0; j < 5; j++) {
                  g_total += g_sum[j];
            }
      }
}
```

Some tasks cannot always be broken up to run on many processors. This problem is demonstrated in the last part of the above program where it makes no sense to break up summation so that it can be done by all the processors. *CPSIM* is flexible enough to be able to vary the number of processors throughout the program. In the above program 5 processors do the initial task and the final summation is done just by one processor.

The Translator

A translator was developed using the UNIX tools Lex (Lexical Analyzer) and Yacc (LALR Parser generator) on a Sun workstation running Version 4.2 BSD UNIX The input to the translator is a program written in *CPSIM* as described above. This section describes the actions performed by the translator-augmenter in creating a trace generator.

One of the tasks of the translator is to add the necessary declarations and definitions such as the PROC and the CYCLE[P] variables. PROC is defined as a global variable and keeps information on the currently executing processor. CYCLE[P] is a global array which keeps track of current machine cycle of Processor P.

The parser is used to construct a symbol table which stores information about the variables and arrays stored in shared memory such as their sizes, starting location and array dimension. Storage management is made in the order of declaration. We assume a linear memory organization. The symbol table is an important aid in determining the address of a shared-memory variable which is being accessed by the program.

The translator also inserts statements to increment the machine cycle (CYCLE[P]) after the execution of an instruction and includes Print Statements at appropriate points to reflect a shared-memory access. These are the two most important type of statements which are included into the trace generator. The way these new statements are added to the trace generator is very logical. Each type of operation such as an addition or an assignment takes a constant number of machine cycles to execute. Based on this model and given the number of the processor which is currently executing we increment the number of cycles spent by that processor for the given statement or set of statements. The time taken to execute a function call is a factor dependent on the number of its arguments, how they are passed and the time required to build the activation stack. In this model however, we assume that it takes a constant amount of time to build an activation stack. The number of machine cycles for the execution of each instruction of a commercial microprocessor is specified in the file "MP.DATA" a portion of which is shown below:

```
#define    mp_add    1    /* ADD Instruction consumes 1 cycle */
#define    mp_sub    2    /* SUB Instruction consumes 2 cycles */
#define    mp_mm     1    /* DECREMENT operator consumes 1 cycle */
```

This file is set up and changed by the user to define the processor to be simulated.

Whenever a variable stored in shared memory is accessed for reading or writing the translator determines its location in shared memory by accessing the symbol table. A print statement is included in the trace generator which prints out the machine cycle, processor number and the memory location just accessed. The statement is of the form:

```
print(cycle, processor, memory location);
```

There are certain occasions however, when a shared memory variable is accessed and causes a print statement to be generated even when traces should not be generated. This problem commonly occurs when a function call is made in a region of the program for which memory traces are not being generated i.e outside of a *pardo* loop. To circumvent this problem the processor number is set to a negative value (-2) so that such traces can be detected before they are generated. The *print* function checks for legal processor identification before printing the trace and takes care of this problem.

A major problem is to be able to pass information about the current cycle, processor and addresses of variables and arrays to functions since variables declared within a function are local to it and do not reflect the addresses of shared memory variables which have been declared in the main program and stored in the symbol table. Information such as the processor identification and cycle is passed to the function by implementing these variables as global variables thus making them accessible to all parts of the program. The Address of a shared memory variable is passed to a function by making modifications to the function call and the formal parameter definitions and declarations. The example below demonstrates how the translator modifies the function head in order to pass the address of a variable which has been passed by reference to the function.

The function Swap requires parameters which are passed by reference

```
/* function to swap two different arrays in memory */

swap(g_array1, g_array2)
int g_array1[];
int g_array2[];
{
      Body
}
```

The Modified function as generated by the translator is given below:

```
/* function to swap two different arrays in memory */

swap(g_array1, g_array1_base, g_array2, g_array2_base)
int g_array1[], g_array1_base;
int g_array2[], g_array2_base;
{
      Body
}
```

The original and modified function calls are shown below:

```
swap(&val1, &val2); is modified to

swap(&val1, val1_base, &val2, val2_base);
```

The additional variables *g_array1_base* and *g_array2_base* have been added to the program and are used to pass the address of the referenced variable. They are also assigned values in the symbol table where they are defined.

Other code that is included into the trace generator is for the initialization of newly declared variables, code to calculate the clock cycle when processors are synchronized and code for supportive functions such as the print function.

The traces generated by the trace generator are statically obtained. We do not take into consideration the effects of the memory contention, i.e., delays in memory access times. An user of these traces needs to know the points where the delays need to be incorporated in order to ensure an accurate trace. For this reason the trace generator incorporates breakpoints in the program where these delays must be included by the user to synchronize the processors. The need for synchronization arises due to two factors. Firstly, data dependencies within the program dictate that the processors synchronize. Secondly, the delays due to memory contention need to be included into the trace at some logical stage. The breakpoint is this logical stage where data dependencies and delays can be handled. It is indicated by an invalid processor identification of (-1). The breakpoint is generated at the end of the *pardo* loop and has the following form:

cycle number (max), processor number (-1) and memory location (0)

The cycle number at this point is the maximum cycle reached by any processor at this stage. The trace generator obtained has a routine which updates the cycles of all processors at this point. When a (-1) processor identification is noticed a simulation of the memory contention using the traces generated till this point is carried out and delays are incorporated into the traces thus generating the accurate trace and synchronizing all processors by updating them with the latest machine cycle.

Example

Given below is an example of a parallel program which performs matrix multiplacation and its trace-generator.

```
1  #define     N 3
2  #define P 3
3
4   main( )
5   {
6          int i, j, k, sum;
7          int G_Prod[N][N];
8          int G_Mat1[N][N];
9          int G_Mat2[N][N];
10
11         pardo (i = 0; i < P; i++) {
12           for (j = 0; j < N; j++) {
13               sum = 0;
14               for (k = 0; k < N; k++)
15                   sum = sum + G_Mat1[i][k] * G_Mat2[k][j];
16               G_Prod[i][j] = sum;
17           }
18         }
19  }
```

The variables defined in line 4 are local variables local to each processor. Lines 5-7 define three arrays which are stored in global memory. Line 9 demonstrates the use of a *pardo* loop in which 3 processors execute the code from lines 10-15.

Given below is a trace generator for the above matrix-multiplacation program. Lines 1-5 are definitions which have been added to the original program. P defines the number of processors, PROC defines the currently executing processor and CYCLE[P] keeps track of the current cycle. Lines 13-16 initialize the cycles and line the Processor value PROC is initialized in line 19. Statements such as those on lines 21, 22 and 24 increment the machine cycles for a particular processor depending on the operation performed. Line 29 symbolizes a print statement for a global variable which is accessed on line 35. It prints the machine cycle, processor number and the location of the variable. Lines 44-50 demonstrate synchronization due to data dependency. Advantage is taken at this point to include a breakpoint where traces can be updated and this breakpoint is printed on line 52. The code between lines 44-50 calculates the maximum time taken by any processor. This code is not very useful here but is important when other *pardo* loops follow i.e., when other sections of the program which are dependent on the previous sections for execution. Finally, lines 59-66 show the addition of a *print* function.

```
1   #define P 3
2   /* Included Cycle Variable */
3   int CYCLE[P];
4   /* Included Processor Register */
5   int PROC;
6   main( )
7   {
8           int i, j, k, sum;
9           int G_Prod[3][3];
10          int G_Mat1[3][3];
11          int G_Mat2[3][3];
12
13          /* Initialize cycles */
14
15          for (PROC = 0; PROC < P; PROC++)
16                  CYCLE[PROC] = 1;
17
18          for (i = 0; i < 3; i++) {
19                  PROC = i;
20
21                  CYCLE[PROC] += 1;
22                  CYCLE[PROC] += 2;
23                  for (j = 0; j < 3; j++) {
24                          CYCLE[PROC] += 1;
25                          sum = 0;
26                          CYCLE[PROC] += 1;
27                          CYCLE[PROC] += 2;
28                          for (k = 0; k < 3; k++)
29                          print(CYCLE[PROC], PROC, i*k + 9);
30                          CYCLE[PROC] += 1;
31                          print(CYCLE[PROC], PROC, k*j + 18);
32                          CYCLE[PROC] += 1;
33                          CYCLE[PROC] += 3;
34                          CYCLE[PROC] += 3;
35                          sum = sum + G_Mat1[i][k] * G_Mat2[k][j];
36                          print(CYCLE[PROC], PROC, i*j + 0);
37                          CYCLE[PROC] += 1;
38                          CYCLE[PROC] += 1;
39                          G_Prod[i][j] = sum;
40                  }
41                  CYCLE[PROC] += 3;
42
```

```
43          }
44          /* synchronization: bring cycles up to date */
45          {
46                  int max = 0;
47                  for (PROC = 0; PROC < P; PROC++)
48                          if (CYCLE[PROC] > max ) max = CYCLE[PROC];
49                  for (PROC = 0; PROC< P; PROC++)
50                          CYCLE[PROC] = max + 1;
51
52                  print(CYCLE[PROC], -1, 0);
53
54          }
55
56  }
57
58
59  /* Print Function */
60
61  print( cycle, proc, memloc)
62  int cycle, proc, memloc;
63  {
64          if (proc >= -1)
65                  printf("%d, %d, %d0, cycle, proc, memloc);
66  }
```

Conclusions & Scope For Further Research

The modified 'C' language and the translator provide a very effective means of emulating a shared memory multiprocessor system in the absence of such a system. The traces generated by the trace generator can be used to further simulate the memory contention of such systems among other possible simulations. We have also provided a framework by which traces for different types of processors can be generated. Such traces can be compared and used in making a decision on what microprocessor can be effectively used to build a parallel system. At the *University of Pittsburgh* we used this tool to generate traces for various numbers of Processors and memory sizes for a study of memory contention in an optically addressed memory scheme (Chiarulli et al. 1987).

One problem which remains to be solved is that of synchronization. In our model (MIMD) of computation we have assumed that the programmer would supply the step in the program where all processors would have to be synchronized with respect to each other. We need to include other synchronization primitives so that different processors need not execute a similar piece of code and and still be able to synchronize with each other. The model does not implement locks and semaphores and we will consider these features in future research. Locks and Semaphores can be implemented by performing a discrete event simulation of the parallel loops. Finally, the 'C' Subset is still in a very primitive form. More advanced features can be added so that the programmer is not constrained to writing programs in a very small language.

References

Chiarulli, D.; Melhem, R.; and Levitan, S. 1987. "Using Coincidental Optical Pulses for Parallel Memory Addressing." *IEEE Computer*, no 12 (Dec), Vol 20: 48-57.

Gottlieb, A.; Grishman, R.; Kruskal, C.; and McAuliffe, K.; 1983. "The NYU Ultracomputer- Designing an MIMD Shared Memory parallel computer", *IEEE Transactions on Computers* (Apr): 175-189.

Wulf, W.A.; and Bell, C. G; 1972 "C.mmp -A multimicroprocessor", *AFIPS Conference Proceedings, FJCC* Vol 41: 765-777.

Patel, Janak. H; 1981 "Performance of Processor Memory Interconnections for Multiprocessors", *IEEE Transactions on Computers*, Vol c-30 (Oct): 771-780

Lee, Ronald.; Yew, PenChung.; Lawrie, Duncan; 1987 "Data Prefetching in Shared Memory Multiprocessors", *Proc. of 1987 Conference on Parallel Processing*, :28-31

Smith, B.J.; 1978, "A Pipelined Shared Resource MIMD Computer", *Proceedings of 1978 International Conference on Parallel Processing*, :6-8.

Reeves, Anthony.; and Bergmark, Donna.; 1987 "Parallel Pascal and the FPS Hypercube Supercomputer", *Proceedings of 1987 International Conference on Parallel Processing*, :385-388.

Dennis, J.B.; 1980 "Data Flow Supercomputers", *IEEE Computer*, Vol 13, (Nov): 48-56.

Bain, W.L.; and Ahuja, S.R.; 1981 "Performance Analysis of High Speed Digital Buses for Multiprocessing Systems.", *Proc. of 8th Annual Symposium on Computer Architecture*, (May): 107-131.

Chang, D.; Kuck, D.J.; and Lawrie, D.H.; 1977 "On the Effective Bandwidth of Parallel Memories", *IEEE Transactions on Computers*, Vol c-26, no 5 (May): 480-490.