一個在 Windows NT 上的多引線翻譯器
A Multithreaded Translator for Parallelizing Compilers on Windows NT*

蔡定谷 楊朝棟 曾憲雄

Ting-Ku Tsai
國立交通大學資訊科學系
Dept. Comp. & Info. Sci.
Nat'l. Chiao Tung Univ.
Hsinchu, Taiwan 300, ROC

Chao-Tung Yang†
行政院國家太空計畫室籌備處
地面系統計畫
ROCSAT Ground Segment
Nat'l. Space Program Offi.
Hsinchu, Taiwan 300, ROC

Shian-Shyong Tseng‡
國立交通大學資訊科學系
Dept. Comp. & Info. Sci.
Nat'l. Chiao Tung Univ.
Hsinchu, Taiwan 300, ROC

摘要

多重引線的技術對許多領域是非常地重要的，例如：平行編譯器、即時多媒體系統、分散式系統。基於上述的概念，我們在 Windows NT 上製作了一個能將單一引線行程轉換成多重引線行程的翻譯器，並且在兩顆 CPU 的多處理器電腦系統上展示了十個 DOALL 迴圈、兩個 DOACROSS 迴圈、四個 DOCONSIDER 迴圈的效能，這些被轉換成多重引線的應用程式幾乎都能獲得高的 Speedup。同時我們也提出改善執行時期平行化策略的方法，即讓 Inspector 選擇適當的排程，接著 Executor 就使用這個排程，將每個 Wavefront 中的迴圈分割成多重引線。經由實驗結果證明這個新的方法能獲得更高的 Speedup。

Abstract

In this paper we have implemented a translator with translating a single process to multithreads on Windows NT. The performance of ten DOALL loops, two DOACROSS loops and four DOCONSIDER loops are demonstrated on 2-CPU multiprocessor system, and almost these applications translated to multithreads can achieve high speedup rates. Meantime, we have improved the runtime parallelizing strategy by using different schedules in executor phase. The inspector can choose the suitable scheduling strategy and the executor can utilize this schedule to partition loop iterations of a wavefront into multiple threads. By using our translator, the experimental results show that the new method could achieve higher speedup on multiprocessor systems.

1 Introduction

Multithreaded support seems to be the most obvious approach for helping programmers to take the advantage of operating system parallelism. Modern operating systems like Mach, Windows NT [PG95], OS/2, and Solaris all support multiple threads of control within the same address space. With the wide availability of multiprocessor computers in the coming years, multithreaded programming will be a viable and important technique for application programmers to master. With threads, a process can have multiple instruction streams executing simultaneously with much lower overhead than concurrent processes. Threads within the same address space inherently share that memory of process, which makes communication and data sharing among threads efficient. In addition, multithreading technology is absolutely essential for domains such as parallelizing compiler [BEH+94, Wol95, ZC90], real-time multimedia and distributed systems. Although multithreaded is powerful for a lot of multiprocessors, we sometimes still lack good parallelizing compilers to help programmers exploit parallelism and gain performance benefits from parallel machines. Therefore, it has become an important issue to develop parallelizing compiling techniques that exploit the potential power of multiprocessors on multithreaded operating systems. In the past from years, we have designed and implemented a portable FORTRAN parallelizing compiler (PFPC) [YTH+97] with loop partitioning for the AcerAltos 10000 multiprocessor system running OSF/1. The compiler can partition parallel loops into multithreaded codes based on several loop-partitioning algorithms. At the present time, among all the operating systems which support multithreads, Windows NT is the most popular one. Therefore, based upon the past experiences and technologies in building PFPC, we implement a translator with translating single-threaded process to multithreaded process on Windows NT.

This paper describes the processes of implementation of a multithreaded translator to parallelize loops and achieve high acceleration rates on multiprocessor systems. First of all, we present the implementation of an S2M translator with loop partitioning under Windows NT. In order to port the translator to other system environments, a minimal set of thread-related data types and functions on Win32 API is defined and used, which is required for an operating system to support execution of the translator. Besides, this translator is highly modularized, and includes some routines which can be used to generate thread-specific codes and partitioned loops for different platforms; that is, the trans-

*This work was supported in part by NSC of ROC Grant #NSC86-2213-E-009-081.

†E-mail: ctyang@nspo.gov.tw and URL: http://peterson.cis.nctu.edu.tw/ctyang.

‡Corresponding author. Phone: +886-3-5715900 and Fax: +886-3-5721490. E-mail: sstseng@cis.nctu.edu.tw.

lator is portable. This translator can partition DOALL, DOACROSS and DOCONSIDER loops into multithreaded codes based on several loop scheduling algorithms [TN93]. Furthermore, we have improved the run-time parallelizing strategy by using different schedules in executor phase. The inspector can choose the suitable scheduling strategy and the executor can utilize this schedule to partition loop iterations of a wavefront into multiple threads. The experimental results clearly show that our translator achieves good speedup under Windows NT. In the study of high-performance parallelizing compilers, results of this paper will be able to deliver theoretical and technical contributions.

2 Technologies used in Our Translator

2.1 Windows NT

In Windows NT, a *process* is a collection of system resource like memory address space, file and device handles, security attributes, synchronization objects,plus at least one *thread of execution*. Each *thread* in a process also has private resources: kernel and user stacks, a set of registers and object attributes, and a program counter. Thread in a process access and share all the resources of the process, for example, all threads can access the same memory space. A process is a Windows NT object with several properties: security attributes, execution context, scheduling priority, and processor affinity; these properties affect all threads running in the process's address space.

2.1.1 Thread Scheduling

The scheduling policy of Windows NT is *preemptive multitasking*. Unlike DOS or Microsoft Windows, Windows NT is truly a multitasking environment because it allows many processes and applications to be run concurrently. Each program believes it has the whole machine to itself; however, the actual hardware is limited to the number of processors, the available physical memory, and other system resources. In order to emulate a virtual machine for each process, Windows NT provides each process with a virtual address space, and it simulates parallelism by dividing processor time among running threads. A unit of time during which a thread can execute on a processor is called a *time slice* or *time quantum*.

Normally, each thread waits for its turn to utilize a processor. When the time quantum of a running thread runs out, that thread is temporarily taken off the processor to allow another thread to execute. The first thread has been preempted. When it is time for the preempted thread to use the processor again, the operating system restores the state of the thread and allows it to continue executing.

2.1.2 Thread Creation

A process begins with a single thread of execution. From this initial thread, other threads can be started by calling `CreateThread`, which has the following interface:

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm,
    DWORD fdwCreate,
    LPWORD lpIDThread );
```

The first argument is a pointer to a `LPSECURITY_ATTRIBUTE` structure that contains a *security descriptor*. The second argument specifies the size of the new thread's stack in bytes. The third argument is a function address from which the new thread begins execution. The fourth argument is data to be passed to this function. The fifth argument specifies whether the thread should begin execution immediately. The last argument is given the thread ID when the thread is created. The `CreateThread` function returns a valid thread handle if it successfully created a thread, or 0 if it failed.

2.1.3 Thread Synchronization

One advantage of multithreaded programming is that you can speed up a program by dividing it into independent threads; the threads can be executed concurrently. For example, suppose we have to program a server application that accepts requests from and provides services to client applications. Using multithreaded programming, we can implement this application with a dispatcher thread that listens to client requests, and then create a separate thread to handle each such request concurrently. The problem with this strategy, however, is that we must properly synchronize the several server threads if they access shared data, in order to ensure that updates do not interfere with one another. In other word, one thread may need to wait for another to finish before it can proceed.

2.1.4 Wait Operations

In Windows NT, a synchronization object can have one of two states: *signaled* (available) or *not-signaled* (owned). A thread wishing to synchronize using one of these objects must call `WaitForSingleObject`, or call `WaitForMultipleObjects` to use a set of them.

```
DWORD WaitForSingleObject (
        HANDLE hObject,
        DWORD dwTimeout
        );
```

`WaitForSingleObject` accepts a synchronization object handle plus a timeout argument. A thread wishing to wait for one or all conditions in a set may call the `WaitForMultipleObjects` function.This function has the following interface:

```
DWORD WaitForMultipleObjects (
        DWORD nCount,
        LPHANDLE lpHandles,
        BOOL bWaitAll,
        DWORD dwTimeout
        );
```

The first argument is the size of the array of object handles, which is passed as the second argument. The third argument is a Boolean flag. If its value is `TRUE`, the function waits for all objects to be in the *signaled* state at the same time. If the flag is `FALSE`, the function returns when

any object in the array attains the *signaled* state. Because of this 'wait for all' behavior, it is easy for this function to block indefinitely if it does not time out, possibly creating system deadlock.

2.2 Comparisons

In OSF/1, Our parallel compiler must use *busy waiting* to wait all threads terminate. The example is stated as follows:

```
/* Main program */ MAIN_()
{
  .....
  for (I=0; I < Count; I++) {
    ThStatus = pthread_create(&Thread,
            pthread_attr_default,
            (void *) FORALL1, &loop[I]);
    pthread_detach(&Thread);
  }
  pthread_mutex_lock(&CountLock);
  while (ThCount != 0)
    pthread_cond_wait(&ThCond, &CountLock);
  pthread_mutex_unlock(&CountLock);
  s_stop("", 0L);
} /* MAIN_ */
void FORALL1(loop)
struct loop_args *loop;
{
  .....
  pthread_mutex_lock(&CountLock);
  ThCount--;
  pthread_mutex_unlock(&CountLock);
  pthread_cond_signal(&ThCond);
}
```

In Windows NT, our parallel compiler can use object (function) which is supported by Win32, and it is straightly controlled by NT operating System. Therefore, the programmer can simplify the code. The example is stated as follows:

```
void main(void)
{
  .....
  for (I=0; I < Count; I++) {
  ThStatus[I]=CreateThread(NULL, 0,
       (LPTHREAD_START_ROUTINE)FORALL2,
       (LPVOID)&loop[I],0,(LPVOID)&ThreadID);
  }
  WaitForMutipleObjects(Count,ThStatus,
                TRUE,INFINITE);
}
```

OSF/1 provides P Threads and C threads packages. The threads implementation must provide two synchronization methods: *mutex objects* for short-duration mutual exclusion, and *condition variables* for event notification. Win32 of Windows NT provides two synchronization objects: `WaitForSingleObject` and `WaitForMultipleObjects`. We use `WaitForMultipleObjects` to implement synchronization.

3 The Translator

3.1 The Overview of PFPC

A portable FORTRAN parallelizing compiler (PFPC) for our shared-memory multiprocessors like the AcerAltos 10000 system, running OSF/1, was designed and implemented at NCTU [YTH+97]. The PFPC generates parallel object codes rather than being just a source-to-source restructurer, and is highly modularized so that porting to other platforms will be very easy. Firstly, the practical parallel loop detector (PPD) proposed is used to test for data dependence relationships and then restructure sequential FORTRAN source programs into parallel forms, i.e., if a loop can be parallelized or partially parallelized, then PPD marks it as a DOALL loop or DOACROSS loop by comment. We implemented the PPD using the auxiliaries of `lex` and `yacc`. PPD takes FORTRAN 77 syntax like programs as input, and yields prompted parallel codes that can be accepted by the f2c directly. Secondly, because there is no FORTRAN compilers for OSF/1 and because multithreading only supports C programming, a FORTRAN-to-C (f2c) converter is used to convert FORTRAN programs output by PPD into their C equivalents. Thirdly, the single-to-multiple threads translator (S2M), takes the program obtained from f2c as input, and generates as output, parallel loops translated into sub-tasks by replacing them with multithreaded codes. Finally, the generated parallel object codes can be scheduled and executed in parallel on multiprocessors to achieve high performance.

3.2 New Strategy for Runtime Scheduling

Before, our executor is implemented using C Thread function calls running under OSF/1. Every wavefront is sequentially executed, and ideally all iterations in the same wavefront are executed in parallel. Nevertheless, if there are too many iterations in the same wavefront, executing all of them simultaneously is impossible. In practice, iterations in the same wavefront are partitioned into equal-sized chunks and every chunk is enclosed in one thread, the threads scheduled by OSF/1 can be executed in parallel. Figure 1 shows wavefronts with sequential inspector. Figure 2 shows wavefronts with parallel inspector.

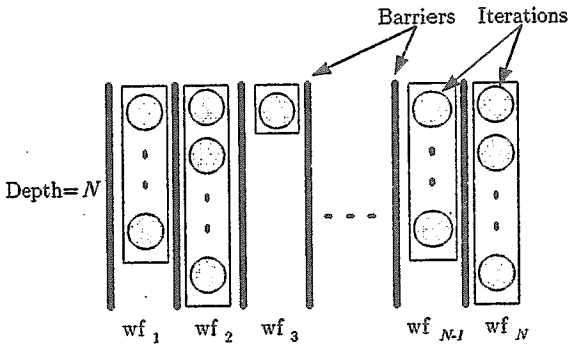

Figure 1: Wavefronts with sequential inspector.

Now, we propose a method to improve the performance of the executor. Firstly, if too many iterations are in the same wavefront, a loop partitioning mechanism (such as

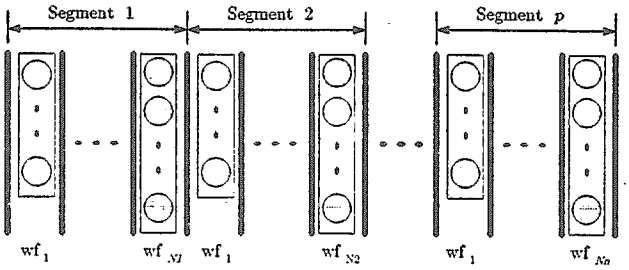

Figure 2: Wavefronts with parallel inspector.

CSS, GSS, Factoring and TSS [TN93]) may be adopted. Secondly, the inspector not only computes number of wavefronts but also depends on the style of loop to choose suitable scheduling strategy. For example, the style of loop detected in inspector is increasing workload, then the inspector may choose TSS or Factoring as scheduling strategy. Thirdly, the executor utilizes the scheduling strategy chosen by the inspector to partition loop iterations of a wavefront into multiple threads. Figure 3 shows a scheduling example of sequential inspector.

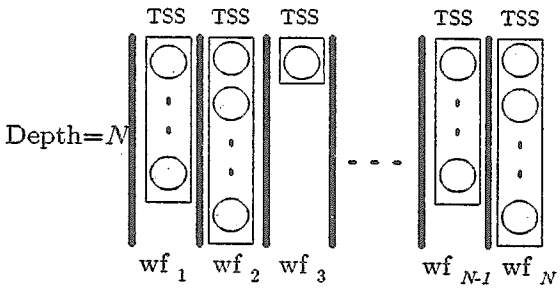

Figure 3: A scheduling example of sequential inspector.

The general output produced by S2M have five sessions. The first session, thread related definition, has some thread related definitions that S2M added to the output. The second session is variables declaration area. The third session partitions the parallel loop according to user assigned loop partition algorithm. The fourth session is just to create threads. The default number of threads to be created is two; however, this number can be changed by a command line option while invoking S2M. The fifth session uses the object for synchronization purpose. We show the main program of the general output produced by S2M as following paragraph.

```
Thread related definition
Variables declaration area
main()
{
    .....
    .....
    Iterations calculation
    Create Thread
    Synchronization
    .....
}
```

We explain our S2M algorithm with general program code. Firstly, we must input a prompted program to our S2M. If the loop is DOALL, the general input code is shown as follows [YTH+97]:

```
main()
{
   Variables declaration area
      .....
   /* DOALL???  */
   for(I=.....)  {
      .....
      .....
   /* L???  */
   }
   /* ENDALL???  */
}
```

Then, S2M will generate the following DOALL thread function [YTH+97]:

```
void DOALL??(loop)
struct loop_args *loop;
{
   int i
   for(i=loop->begin;i<=loop->end;i+=loop->step){
      .....
   }
}
```

If the loop is DOACROSS, the general input code is shown as follows [YTH+97]:

```
main()
{
   Variables declaration area
      .....
   /* DOACR???  */
     for(I=.....)  {
   /* SIGNAL(n) */
      .....
   /* WAIT(n,i-m) */
      .....
   /* L???  */
   }
   /* ENDACR???  */
      .....
}
```

Then, S2M will generate the following DOACROSS thread function [YTH+97]:

```
void DOACROSS??(loop)
struct loop_args *loop;
{
   int i;
   for (i=loop->begin; i<=loop->end; i++) {
      .....
   /* SIGNAL(n) */
      Set ready_n[i] = TRUE
   /* WAIT(n,i-m) */
      Test if the loop iteration is head
      Wait until ready_n[i-m] = TRUE
      .....
   }
}
```

If the loop is DOCONSIDER, the general input code is shown as follows [YTH+97]:

```
main()
{
   Variables declaration area
   .....
   /* DOCNSD???  */
   for(I=.....)  {
      .....
      .....
   /* L???  */
   }
   /* ENDCNSD???  */
}
```

If we use sequential inspector, then S2M will only generate the following EXECUTOR thread function [YTH+97]:

```
Variables declaration area
void main()
{
   .....
   INSPECTOR
   EXECUTOR
   .....
}
void EXECUTOR(loop)
struct loop_args *loop;
{
int i,j;
int INDEX;
for(INDEX=loop->begin;INDEX<=loop->end;INDEX++)
   .....
}
```

If we use parallel inspector, then S2M will generate the following INSPECTOR and EXECUTOR thread functions [YTH+97]:

```
Variables declaration area
void main()
{
      .....
   INSPECTOR
   EXECUTOR
      .....
}
void INSPECTOR(loop)
struct loop_args_insp *loop;
{
int i, ini_index;
int j;
for(ini_index=0;ini_index<=N;ini_index++){
      loop->def[ini_index] = 0;
      loop->use[ini_index] = 0;
}
for(i=loop->begin;i<=loop->end;i++)
      .....
}
void EXECUTOR(loop)
struct loop_args *loop;
{
int i,j;
int INDEX;
for(INDEX=loop->begin;INDEX<=loop->end;INDEX++)
      .....
}
```

4 Experimental Environment and Results

4.1 Environment

Our target machine is two Intel Pentium-133 CPU multiprocessor system, running the Windows NT multithreaded OS that supports Win32 API functions. The system includes 512K external cache and 64MB shared-memory, and a 64-Bit high-speed frame bus. Our translator is coded and compiled by utilizing visual C++ which provides Win32 API functions call. Due to the symmetric architecture, computation tasks can be easily distributed to any available processor. This means that balanced loading of all processors can be achieved.

4.2 Experimental Results

Ten examples are used for DOALL loop parallelization of S2M. Figures 4, 5 and 6 show the speedups of adjoint convolution, Gaussian elimination, matrix multiplication, reverse adjoint convolution, transitive closure, SOR, Jacobi iteration, Gaussian-Jordan elimination, LU decomposition, and all pairs shortest paths by using different program sizes loop. In the experiments, CSS is used to partition every example and obtain their corresponding performances. Obviously, speedup of parallel version is always higher than serial version. So, the S2M translator used in Windows NT can perform high speedup on multiprocessor systems. Particularly, for the loop with uniform workload, such as matrix multiplication shown in Figure 4 (c), it can achieve higher speedup, since the CSS is suitable for the uniform workload loop.

We examine the characteristics of adjoint convolution and reverse adjoint convolution. In Figure 7 (a), because ad-

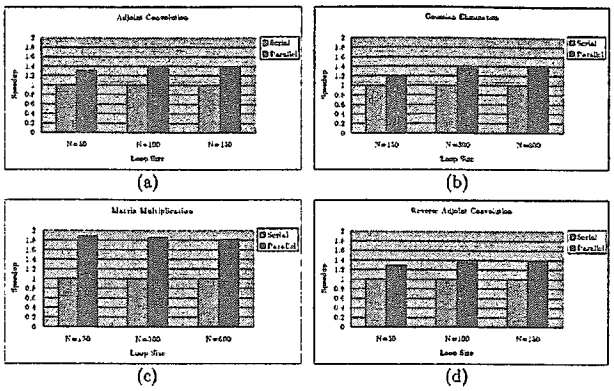

Figure 4: Part I: partial results of DOALL examples.

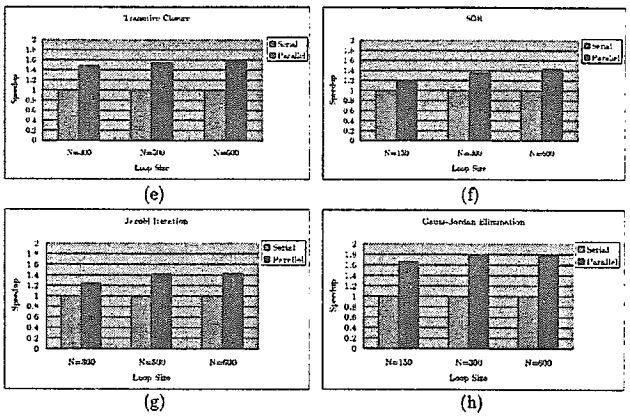

Figure 5: Part II: partial results of DOALL examples.

joint convolution is with decreasing workload, we distribute 1/3 of all workload to the first thread and 2/3 of workload to the second thread. As a result of workload is balanced, the speedup is raised. Moreover, in Figure 7 (b), because reverse adjoint convolution is with increasing workload, we distribute 2/3 of all workload to the first thread and 1/3 of workload to the second thread. As a result of workload is balanced, the speedup is raised. Furthermore, for loop of increasing workload, GSS can distribute workload more balanced, so its speedup is raised again.

In order to compare performances of different loop partition algorithms, we examine five representative applications. Figure 8 shows speedup when applications was run with different loop-partitioning algorithms and arguments. For adjoint convolution and reverse adjoint convolution, Factoring obtains highest performance. For matrix multiplication and transitive closure, CSS/2 obtains highest performance. For Gaussian elimination, TSS obtains highest speedup.

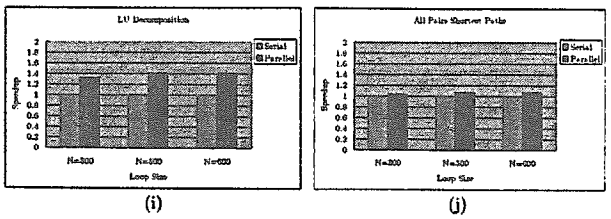

Figure 6: Part III: partial results of DOALL examples.

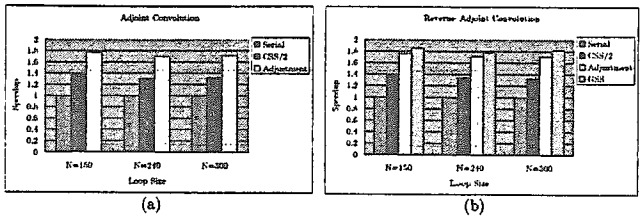

Figure 7: Results of adjusted adjoint and reverse adjoint convolution.

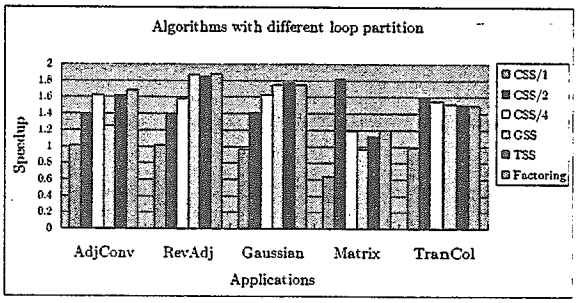

Figure 8: Result of different loop-partitioning algorithms.

We use new runtime parallelizing method to improve the performance of the previous method. Figure 9 shows the speedup of various loop partitions used in executor. Figure 9 (a) is the loop with uniform workload, and parallel inspector can obtain higher performance than sequential inspector. Figure 9 (b) is the loop with increasing workload, and TSS strategy chosen in parallel inspector can obtain highest performance. Figure 9 (c) is the loop with random workload, and TSS strategy chosen in parallel inspector can obtain highest performance. Figure 9 (d) is the loop with non-constant dependence distance and increasing workload. The Factoring strategy chosen in parallel inspector can obtain highest performance.

5 Conclusion and Further Work

In this paper, by calling Win32 API functions of Windows NT, the translator can translate successfully sequential programs to parallel programs with multithreads and run effectively under Windows NT. The performance of ten DOALL loops, two DOACROSS loops and four DOCONSIDER loops are demonstrated on 2-CPU multiprocessor system, and almost these applications translated to multithreads can achieve high speedup rate. As to run-time parallelism, we have proposed a new scheduling approach. A general inspector can choose the suitable scheduling strategy by detecting style of loop in run-time and the executor utilizes this schedule to partition loop iterations of a wavefront into multiple threads. By using our translator, the experimental results show that the new scheduling approach can achieve higher speedup on multiprocessor systems. In the study of high-performance parallelizing compilers, results of this paper will be able to deliver theoretical and technical contributions.

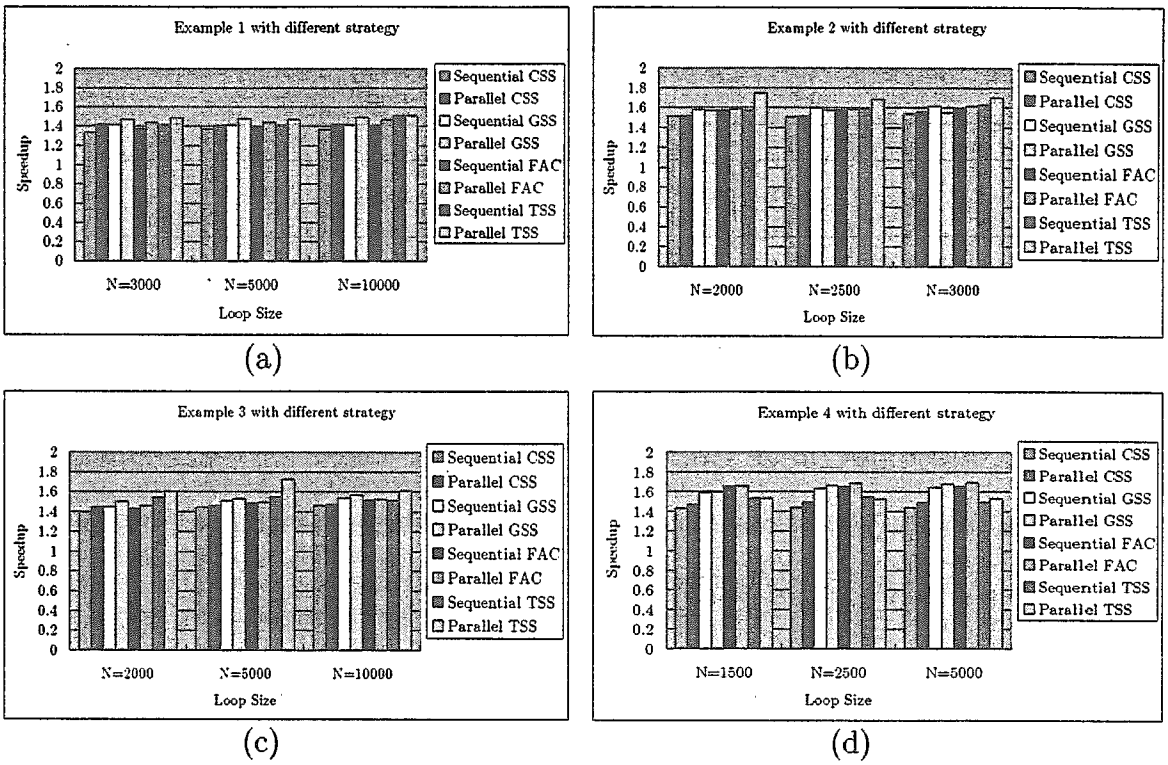

Figure 9: Results of new runtime parallelization.

Most of investigations for parallelizing loops are concentrated on uniform memory access shared-memory multiprocessor systems (UMA). A new research direction is on parallelizing loops for a non-uniform memory access multiprocessor systems (NUMA). In the future, we plan to connect many machines with Windows NT operating system to clustering structure. In this structure, a process with many threads can distribute these threads to many machines and run concurrently in many machines. The clustering structure will exploit the features of distributed shared-memory multiprocessors and eliminate the major interprocess communication overhead for parallelizing compilers.

References

[BEH+94] W. Blume, R. Eigenmann, J. Hoeflinger, and D. Padua, P. Petersen, L. Rauchwerger, P. Tu, "Automatic detection of parallelism: A grand challenge for high-performance computing," *IEEE Parallel & Distributed Technology*, 2(3):37-47, Fall 1994.

[PG95] T. Q. Pham and P. K. Garg, *Multithreaded Programming with Windows NT*, Prentice-Hall, New Jersey, 1995.

[TN93] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87-98, Jan. 1993.

[Wol95] M. Wolfe, *High-Performance Compilers for Parallel Computing*, 137-162, Addison-Wesley Publishing, New York, 1995.

[YTH+97] C. T. Yang, S. S. Tseng, M. C. Hsiao, and S. H. Kao, "A portable parallelizing compiler with loop partitioning," to appear in *Proc. of the NSC ROC (A)*.

[ZC90] H. P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing and ACM Press, New York, 1990.