CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:1165–1176

OpenMP versus threading in C/C++

Bob Kuhn[1], Paul Petersen[1,*] and Eamonn O'Toole[2]

[1]*Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820, U.S.A.*
[2]*Compaq Computer Corporation, Ballybrit Industrial Estate, Ballybrit, Galway, Ireland*

SUMMARY

When comparing OpenMP to other parallel programming models, it is easier to choose between OpenMP and MPI than between OpenMP and POSIX Threads (Pthreads). With languages like C and C++, developers have frequently chosen Pthreads to incorporate parallelism in applications. Few developers are currently using OpenMP C/C++, but they should. We show that converting Genehunter, a hand-threaded C program, to OpenMP increases robustness without sacrificing performance. It is also a good case study as it highlights several issues that are important in understanding how OpenMP uses threads.

Genehunter is a genetics program which analyzes DNA assays from members of a family tree where a disease is present in certain members and not in others. This analysis is done in an attempt to identify the gene most likely to cause the disease. This problem is called linkage analysis. The same sections of Genehunter were parallelized first by hand-threading and then with OpenMP on Compaq Alpha Tru64 systems. We present examples using both methods and illustrate the tools that proved useful in the process. Our basic conclusion is that, although we could express the parallelism using either Pthreads or OpenMP, it was easier to express the parallelism at a higher level of abstraction. OpenMP allowed enough control to express the parallelism without exposing the implementation details. Also, due to the higher level specification of parallelism with OpenMP, the tools available to assist in the construction of correct and efficient programs provide more useful information than the equivalent tools available for hand-threaded programs.

The following concepts are presented:

- **differences between coding styles for OpenMP and Pthreads;**
- **data scoping specification for correct parallel programming;**
- **adapting a signal based exception mechanism to a parallel program;**
- **OpenMP tools: Debuggers—Ladebug, TotalView and Assure; Profilers—Hiprof and GuideView;**
- **performance tuning with memory allocation, synchronization, and scheduling.**

Genehunter does not cover a few important topics in C/C++ programming style, which will be discussed separately. These are:

- **interfacing a GUI team of threads with an OpenMP compute team;**
- **coordinating data structure with scheduling.**

Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: OpenMP for C/C++; Posix Threads; SMP programming techniques; parallel programming tools

*Correspondence to: Paul Petersen, Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820, U.S.A.

Received November 1999
Revised May 2000

1. INTRODUCTION

This paper serves two purposes. First, we present some practical experiences comparing OpenMP [1] to hand-threading using POSIX Threads (Pthreads) [2] in a C program called Genehunter. The general availability of both Pthreads and OpenMP implementations for C/C++ makes this comparison interesting. Second, with respect to the internal techniques of OpenMP, we use this case to illustrate some threading issues that we think are important for OpenMP implementers and researchers.

Why use OpenMP when one can achieve the same thing with Pthreads directly? We have found that there is a difference between writing a code from scratch for parallelism, and retrofitting it into an existing code. Almost any parallelism system looks good when the code is designed for that system. It is much harder to take a serial program and maintain its structure when introducing parallelism. OpenMP excels at the latter.

The same sections of Genehunter were independently parallelized by hand-threading and with OpenMP on Compaq Alpha Tru64 systems. We present both ways of parallelizing the program and the tools that proved useful in the process. Our basic conclusion is that hand-threading can be used to implement anything in an OpenMP runtime library, but why should the application developer need to reinvent efficient implementations of the most commonly used operations in high performance computing for every application.

2. GENEHUNTER

2.1. What it does and how

Genehunter is a genetics program that analyzes a set of DNA assays from members of a family tree. This analysis is done in an attempt to identify the gene most likely to cause a disease based on its presence in certain members of the family and not in others. This problem is called linkage analysis [3–5].

The backbone of Genehunter is the very rapid extraction of complete multipoint inheritance information from pedigrees of moderate size. Quick calculations involving dozens of markers, even in pedigrees with inbreeding and marriage loops, is possible. The multipoint inheritance information allows the reconstruction of maximum-likelihood haplotypes for all individuals in the pedigree and information content mapping. The information content mappings measure the fraction of the total inheritance information extracted from the marker data.

The program computes the likelihood that certain markers on the gene are associated with the disease. The essential method of Genehunter is to construct a hidden Markov model representing the likelihood of all possible ‘inheritance vectors’, a way of representing inheritance of the disease through the family tree.

2.2. Parallelism in the method

The number of inheritance vectors grows exponentially with the size of the family tree, and therefore there is not only interest in parallel processing with respect to making the analysis faster but also to use large amounts of shared memory for the large number of inheritance vectors.

The majority of the parallelism is in the maximum likelihood computation, which uses FFTs in treating the matrix computation as a convolution.

3. OPENMP VERSUS HAND-THREADED PARALLELISM

3.1. Code fragment from Genehunter

Compute bound parallelism for shared memory parallel machines is frequently implemented the same, from the application developer's point of view, in both Pthreads and OpenMP. Here we compare the basic parallelization points in the hand-threaded and the OpenMP versions of Genehunter.

In the OpenMP version, shown in Figure 1, one `parallel` region is defined which contains one worksharing `for` loop. By default, all variables other than those specified in the private clause are shared. In OpenMP C/C++, the block that defines the `parallel` region sets the context for additional private variables. In this example, two private pointers, `affected_allele` and `local_my_famtree`, are declared in this block. The `firstprivate` clause can be used to initialize per-thread variables. This is sometimes used to enhance cache performance or to initialize temporary variables. Rather than having all threads reference shared variables, local copies are made. In the OpenMP `for` pragma, the `lastprivate` clause is used to copy out the last index value of the loop for later use. Two sum reductions occur (on the variable `exp` and `exp_2`). For serial compilation, the pragmas can simply be ignored.

Although the hand-threaded version implements similar parallel activity, the modifications to the serial code are much larger. The parallel region must be taken out-of-line and put into a separate subroutine. A data structure containing all the private information for each thread must be created. This data structure is initialized in the routine that contains the parallel region and it is unpacked in the out-of-line routine. For the parallel region in Figure 1, this structure has over 20 elements. Code is constructed to assign lower and upper bounds for the iterations of the parallel for loop that each thread will execute, and the loop must be tuned by hand; there is no schedule parameter which can be easily changed. OpenMP also avoids the need to create and fill an array of local contributions to the sum reductions, which are then added by the master thread upon completion of the out-of-line routine.

Transformations to express the hand-threading do not include the utility subroutines and data structure needed to manage the threads. Utility routines are needed to create and destroy thread teams, point to and execute the out-of-line routine, and to manage mutexes. These additional routines are layered on top of Pthreads to manage the data structures for thread pointers and synchronization operations. These routines also provide for portability between threading packages. This thread utility package is put into a separate file and made platform-specific.

3.2. Data scoping specification for correct parallel regions

Most of the work in parallelizing a section of code for SMP is involved in determining the set of variables that are shared and the set of variables that are private. This work can be thought of as making the data accessed in the parallel region thread-safe.

Hand-threading requires placing all variables that are to be private into a data structure and passing a pointer to each thread's private copy down the call tree. It is necessary to do the same with shared

```
#pragma omp parallel private(i,j,dis_geno,num_aff,prob) \
    firstprivate(vec, uninformative_mask)
{
    int **affected_allele = NULL;
    MY_PED_MEMBER *local_my_famtree;

    local_my_famtree = my_famtree;
    matrix (affected_allele, num_in_ped, 2, int);

    #pragma omp for nowait schedule(guided) lastprivate(new_vec) \
        reduction(+ : exp, exp_2)
    for (new_vec=0; new_vec < real_num_vecs; new_vec++) {

        /* Check to see if ctl-C was pressed. */
        if (has_hit_interrupt())
            continue;

        /* now every member of famtree has two placeholder alleles
            filled in and we can advance to the prob assignment
            for each of the markers */
        for (i=0; i < num_in_map_order; i++) {
            prob = graph_assign_probs(local_my_famtree, vec, originals,
                real_non_originals, map_order[i], NULL, FALSE);
            pvector[i][new_vec] = prob;
        }

        /* now obtain NPL score for this assignment of placeholder
            alleles for this pedigree */
        for (i=0,num_aff=0; i < num_in_ped; i++) {
            if (allele_data[i][0] == 2) {
                /* this individual is affected */
                affected_allele[num_aff][0] =
                    local_my_famtree[i].place_allele[0];
                affected_allele[num_aff][1] =
                    local_my_famtree[i].place_allele[1];
                num_aff++;
            }
        }
        score[new_vec] = apm_score(affected_allele, num_aff);
        exp += score[new_vec];
        exp_2 += (score[new_vec]*score[new_vec]);
    }
    unmatrix (affected_allele, num_in_ped, int);
}
/* Check to see if ctl-C was pressed. */
has_hit_interrupt();
```

Figure 1. Genehunter code fragment showing how OpenMP pragmas are used.

variables because they may be allocated in a stack frame entered before the parallel region is executed. For threads to access these shared variables a data structure needs to be created and a pointer passed. Since structures need to be created either way, for read-only shared data the hand-threaded version used a mechanism similar to first private, where a separate copy of the shared data was created for each thread. Overall, the changes to the serial version were significant.

It was easier to use OpenMP. A combination of private, shared and reduction clauses were used with the thread private pragma. Very few changes to the serial version were required.

3.3. Adapting a signal based exception mechanism to a parallel region

Something that occurs more with C/C++ applications than with Fortran applications is that the program uses a sophisticated user interface. Genehunter is a simple example where the user may interrupt the computation of one family tree by pressing control-C so that it can go on to the next family tree in a clinical database about the disease. The premature termination is handled in the serial version by a C++ like exception mechanism involving a signal handler, setjump, and longjump.

OpenMP does not permit unstructured control flow to cross a parallel construct boundary. We modified the exception handling in the OpenMP version by changing the interrupt handler into a polling mechanism. The thread that catches the control-C signal sets a shared flag. All threads check the flag at the beginning of the loop by calling the routine `has_hit_interrupt()` and skip the iteration if it is set. When the loop ends, the master checks the flag and can easily execute the longjump to complete the exceptional exit (see Figure 1).

Implementing a hand-threaded version of this modified exception mechanism has not been done because of the complexity in the hand-threaded code, but the polling technique could be implemented.

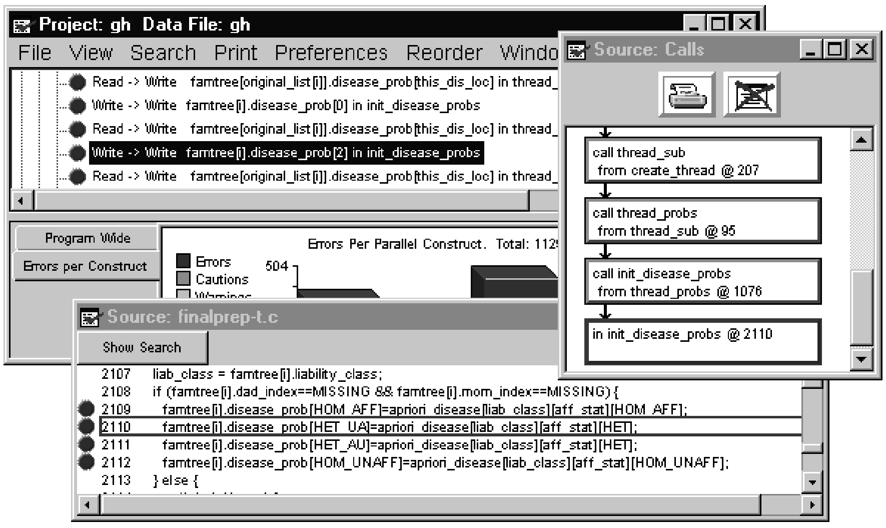
4. OPENMP VERSUS HAND THREADING TOOLS

4.1. Debugger versus behavior analyzers

A multithreaded debugger can be used to examine both the hand-threaded and the OpenMP version on Tru64 Unix. The Tru64 Ladebug debugger and the TotalView debugger [6] are OpenMP aware in that they display directives in source code within the debugger. However, using the debugger requires the user to set good breakpoints, switch between threads and examine the dynamic state of the data structures for incorrect data, and mentally trace back to the source of the bad data. Crucial timing between threads is difficult to reconstruct.

Assure, in the KAP/Pro Toolset [7], takes a different approach. It runs an instrumented copy of the program which traces where multiple threads touch the same virtual memory location. When that occurs, it attempts to explain why that may be valid in the parallel-programming model. If no explanation is found, it records the parallelism defect in a diagnostic database and continues execution of the program. We ran Assure on both the hand-threaded and the OpenMP version of Genehunter.

Figure 2 is a screen captured from the Assure for Threads run on the hand-threaded version of Genehunter. This figure shows a fragment of the source code referred to in the report. The report shows that it is possible for more than one thread to modify or access the elements of the shared `famtree[].disease_prob[]` vector in the routine `init_disease_probs()` at lines `2110`


Project: gh Data File: gh
File View Search Print Preferences Reorder Windo
Source: Calls
Program Wide
Errors per Construct
Errors Per Parallel Construct. Total: 112
Errors
Cautions
504
Source: finalprep-t.c
Show Search

```
Read -> Write   famtree[original_list[i]].disease_prob[this_dis_loc] in thread_
Write -> Write  famtree[i].disease_prob[0] in init_disease_probs
Read -> Write   famtree[original_list[i]].disease_prob[this_dis_loc] in thread_
Write -> Write  famtree[i].disease_prob[2] in init_disease_probs
Read -> Write   famtree[original_list[i]].disease_prob[this_dis_loc] in thread_
```

```
call thread_sub
 from create_thread @ 207
call thread_probs
 from thread_sub @ 95
call init_disease_probs
 from thread_probs @ 1076
in init_disease_probs @ 2110
```

```
2107   liab_class = famtree[i].liability_class;
2108   if (famtree[i].dad_index==MISSING && famtree[i].mom_index==MISSING) {
2109     famtree[i].disease_prob[HOM_AFF]=apriori_disease[liab_class][aff_stat][HOM_AFF];
2110     famtree[i].disease_prob[HET_UA]=apriori_disease[liab_class][aff_stat][HET];
2111     famtree[i].disease_prob[HET_AU]=apriori_disease[liab_class][aff_stat][HET];
2112     famtree[i].disease_prob[HOM_UNAFF]=apriori_disease[liab_class][aff_stat][HOM_UNAFF];
2113   } else {
```

Figure 2. Assure for Threads report showing data race conditions involving vector `famtree[].disease_prob[]`.

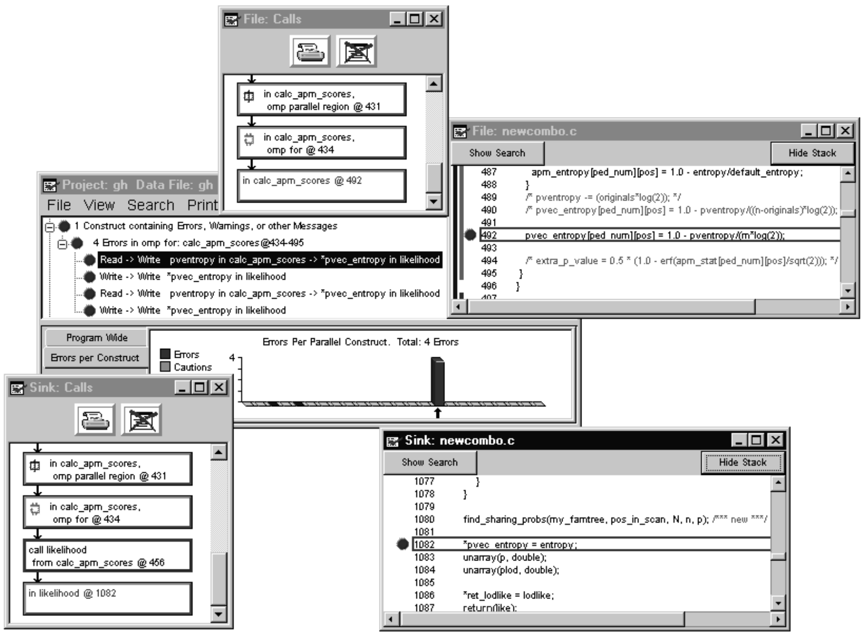
through `2112`. This diagnostic message means that the final value of this variable would be ambiguous depending on which thread wrote the last value.

This pattern of data races in the diagnostic messages indicates that the variable structure `famtree[].disease_prob[]` should be made private to each thread. Making this change in the hand-threaded version is difficult: (i) a local copy of this structure must be defined; (ii) it must be allocated and freed in multi-threaded mode; (iii) the pointer must be passed down through to call tree to these routines; (iv) finally the source code must be modified to use the local copy.

One of the advantages of OpenMP is that its parallel programming model is much more structured than with hand-threading, so that Assure's explanation mechanism is stronger. We were confident that if Assure for OpenMP did not record any diagnostics, the parallel version would produce the same results as the serial version. Assure for Threads is weaker. It is only capable of detecting unsynchronized, shared accesses by multiple threads.

The report from the Assure for OpenMP run on Genehunter, shown in Figure 3, indicates a problem with the double precision variable `pventropy`. The highlighted diagnostic indicates that several threads can write a value for this variable in routine `likelihood()` at line `1082` in an arbitrary order. The highlighted diagnostic indicates that consequently the thread that is reading the variable `pventropy` in the routine `calc_apm_scores()` at line `492` will get an ambiguous value.

As in the case above, a private copy of the variable `pventropy` should be created. This is much easier when using OpenMP. The variable `pventropy` need only be added to the private list.



in calc_apm_scores,
omp parallel region @ 431

in calc_apm_scores,
omp for @ 434

in calc_apm_scores @ 492

1 Construct containing Errors, Warnings, or other Messages
4 Errors in omp for: calc_apm_scores@434-495
Read -> Write pventropy in calc_apm_scores -> *pvec_entropy in likelihood
Write -> Write *pvec_entropy in likelihood
Read -> Write pventropy in calc_apm_scores -> *pvec_entropy in likelihood
Write -> Write *pvec_entropy in likelihood

```
487          apm_entropy[ped_num][pos] = 1.0 - entropy/default_entropy;
488        }
489        /* pventropy -= (originals*log(2)); */
490        /* pvec_entropy[ped_num][pos] = 1.0 - pventropy/((n-originals)*log(2));
491
492        pvec_entropy[ped_num][pos] = 1.0 - pventropy/(m*log(2));
493
494        /* extra_p_value = 0.5 * (1.0 - erf(apm_stat[ped_num][pos]/sqrt(2))); */
495      }
496    }
```

in calc_apm_scores,
omp parallel region @ 431

in calc_apm_scores,
omp for @ 434

call likelihood
from calc_apm_scores @ 456

in likelihood @ 1082

```
1077          }
1078      }
1079
1080      find_sharing_probs(my_famtree, pos_in_scan, N, n, p); /*** new ***/
1081
1082      *pvec_entropy = entropy;
1083      unarray(p, double);
1084      unarray(plod, double);
1085
1086      *ret_lodlike = lodlike;
1087      return(like);
```

Figure 3. Assure for OpenMP report showing data race conditions involving the variable `pventropy`.

We found that Assure could be used to parallelize the OpenMP version by speculatively putting in the OpenMP pragmas and letting Assure tell us where the problems existed. To use speculation in adding parallelism with only a debugger would be almost useless.

Compaq has a new product called Visual Threads which is available starting with Tru64 Unix V4.0D. This tool can be used to debug threaded applications by looking at runtime behavior. It can also provide some performance analysis. Although it seems quite powerful, because it is new it has not been tried on the hand-threaded version of Genehunter.

4.2. Profiling versus GuideView

When executing the hand-threaded version of Genehunter, we found that it was running so slowly that it appeared to have an infinite loop. Rather than spending the time to track this problem down with a profiler, we switched over to the OpenMP version. GuideView (a component of the KAP/Pro Toolset [7]) has the ability to integrate data from multiple runs at the same time. Seeing one, two, three, and four processor runs on each parallel region at the same time makes it very easy to detect bottlenecks.

With hand-threading, the only choice is a multithreaded profiler. The multithreaded profiler Hiprof on Tru64 Unix produces one profile per thread. This can provide useful information but it can be awkward to integrate information from several threads. Hiprof is an atom tool that produces an

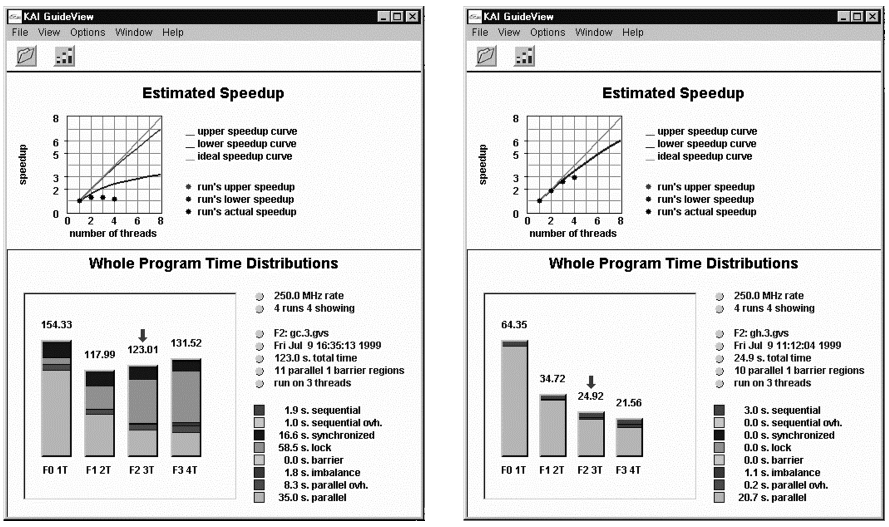


Figure 4. Genehunter performance before tuning (left) and after tuning (right).

instrumented and somewhat slower executable. Profilers are most useful for analyzing the serial parts of the code before parallelization because they show which paths through the code are frequently executed.

In the OpenMP version of Genehunter, we used the GuideView performance analysis tool to determine that there was excessive critical section time around the memory allocation calls. The left screenshot, in Figure 4, shows runs on one, two, three, and four threads. Although the parallel overhead for the one thread run is not negligible, it grows to dominate the running time. In fact, for more than two threads, the wall-clock time increases despite the fact that the productive work appears to be parallelized correctly! The biggest portion of that overhead is in lock time—the time one thread spends waiting for other threads to clear the critical section. To find out where the lock time is coming from, the GuideView user can view for all the parallel regions in Genehunter sorted by overhead.

We fixed the problem with a multithreaded memory allocation routine. When we reran the modified executable with GuideView, the lock time was virtually eliminated and the apparent infinite loop disappeared. The right screenshot, in Figure 4, shows Genehunter performance after this improvement and with the best scheduling options.

5. COMPETING WAYS TO DO FINE GRAIN PARALLELISM

5.1. Scheduling

One of the advantages of OpenMP we found was the simplicity of modifying the scheduling of work. In the hand-threaded version, work distribution must be directly coded in the application. In the OpenMP

```
/* Worker waits on master to give start signal */

    status = pthread_mutex_lock(&start_mutex);
    check(status, "Start_mutex lock bad status in worker\n");

    while (thread_arg_t->proceed) {
        if (thread_exit) {
            /* printf("worker %i exits\n", my_num); */
            /* fflush(stdout); */
            pthread_mutex_unlock(&start_mutex);
            pthread_exit(&my_num);
        }
        status = pthread_cond_wait(&start_cond_var, &start_mutex);
        check(status, "Start_cond_wait bad status in worker\n");
    }

    status = pthread_mutex_unlock(&start_mutex);
    check(status, "Start_mutex unlock bad status in worker\n");

/* Master signals when to start */

    /* Lock mutex */
    status = pthread_mutex_lock(&start_mutex);
    check(status, "Start_mutex lock bad status in main\n");

    /* Lower predicate on threads that are to be allowed work */
    for (worker_num = 0; worker_num < *num_workers; worker_num++) {
        (thread_arg+worker_num)->proceed = 0;
    }

    /* Wake all threads*/
    pthread_cond_broadcast(&start_cond_var);

    /* Unlock mutex */
    status = pthread_mutex_unlock(&start_mutex);
    check(status, "Start_mutex unlock bad status in main\n");
```

Figure 5. Genehunter hand-threaded code implementing a barrier.

version, we started with dynamic scheduling. Performance analysis (GuideView) showed us that there was little load imbalance. However, the roundoff error of reductions was higher than we, as non-experts in genetics, could justify. Scheduling was switched to an evenly distributed static schedule, and accuracy (as measured by reproducible results) increased. The load imbalance increased slightly but GuideView performance analysis shows that the increase was relatively minor.

5.2. Synchronization

OpenMP has an advantage in synchronization over hand-threading where the developer has either to use more expensive system calls than present in OpenMP or to code efficient versions of synchronization primitives. The code in Figure 5 was used to implement a barrier in the hand-threaded

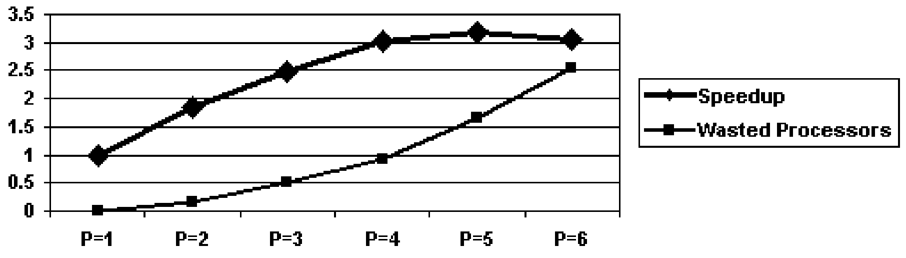


Figure 6. Speedup (top) and wasted processors (bottom) versus number of processors.

version of Genehunter. We discussed and reviewed whether this barrier is as efficient as possible or entirely correct according to the Alpha/Tru64 Unix architecture. In OpenMP, one pragma accesses a well-tuned, well-debugged version.

The code, shown in Figure 5, illustrates how workers are signaled to start their work by the master in the hand-threaded version of Genehunter. All worker threads are sleeping at the `pthread_cond_wait` until the master wakes them. If the current worker thread is among those the master has selected by setting the `proceed` f28.5lag, then the worker is able exit the while loop. Otherwise, it goes back to sleep. To ensure mutual exclusion `start_mutex` is locked and unlocked appropriately by master and worker.

5.3. Performance of Genehunter

Figure 6 shows speedup with a test pedigree on a Compaq GS140 with 8GB RAM and eight 21264 EV6Alpha processors running at 525MHz. The flattening of the curve is to be expected in this version of Genehunter with these data sets. There are two significant parallel regions and one is a parallel sections construct with only two sections. Since the maximum speedup in this region is 2, it starts to limit performance at three and four processors. The hand-threaded version cannot be benchmarked with this data set at this time because this data set was chosen to help debug the multithreaded version.

In hand-threading, the programmer normally forces a thread to sleep when it is not doing productive work. In the Compaq and KAI implementations of OpenMP, the programmer can effect this by a runtime library parameter causing the thread to sleep after a selected number of spin cycles. To illustrate what happens if this parameter is not used, the wasted processors for this job are shown in the lower curve. This is computed first by subtracting the speedup from the number of processors requested by the job to determine the number of idle processors:

- Idle Processors = Processors Requested − Measured Speedup

Because the operating system timeslices a threaded job, it is necessary to adjust the idle processors by the percentage of time processors actually spend in Genehunter.

- Wasted Processors = Idle Processors ∗ CPU Utilization

For this test CPU utilization ranged from 99% for one thread (there is little I/O delay in this test and no other significant jobs competing for the processors) to 86% for six threads. The wasted processors increase non-linearly to two processors for six threads because processors spend considerable time spinning at the region with two sections.

6. SOME IMPORTANT C/C++ TOPICS THAT DO NOT APPEAR IN THIS APPLICATION

6.1. Thread team interactions: GUI versus OpenMP

Some applications have a set of threads for the user interface, which interacts with the team of threads for the computationally intensive part. Threads are serving dramatically different functions. For user interfaces, threads should spend most of their time in sleep mode waiting for events to occur. In the computionally bound parts, threads should be running nearly all the time.

If applications alternate between the computionally bound and the user interface part, then the compute threads need to be put to sleep quickly to avoid wasting cycles. If an application runs the user interface concurrently, signaling between thread teams needs to be managed.

6.2. Efficient parallel region scheduling

We have found that with C/C++ applications more than with Fortran applications the parallel regions are smaller and are executed more frequently. This is because C/C++ is more frequently the choice of interactive or low operations per I/O applications. Therefore, it tends to be very important to:

- keep the threads hot by extending the parallel region to enclose several loops;
- keep cache memory hot by ensuring consistent scheduling between work sharing constructs.

Because dynamic data structures are frequently used in C/C++ applications, multithreaded access to data structures is critical. For linked data structures which are traversed many times, that means that private pointers to different parts of the structure should be computed and retained from pass to pass.

Here OpenMP only has half an advantage. OpenMP runtime scheduling mechanisms are highly tuned and save the hand-threaded developer the work of writing their own efficient scheduling mechanism. However, partitioning the data structures must be done in both models, and the OpenMP static scheduling should be used to ensure that the same work is assigned to the same thread to keep the cache hot.

7. CONCLUSION

We concluded that developing this type of SMP parallelism in C/C++ applications is overall much easier with OpenMP:

(i) OpenMP facilitates making data structures thread-safe, and that is the biggest part of parallelizing an application typically.
(ii) Other features in OpenMP allow the application developer to avoid rewriting tricky threading synchronization. OpenMP is already tuned for speed in computationally bound applications.

(iii) The tools for OpenMP and hand-threading are similar but the OpenMP tools take advantage of OpenMP to simplify parallelizing, debugging, and tuning. Overall, development goes much faster.

REFERENCES

1. OpenMP Architecture Review Board. OpenMP C and C++ application program interface. October 1998. http://www.openmp.org/.
2. IEEE. IEEE P1003.1c-1995: Information Technology–Portable Operating System Interface (POSIX). http://www.ieee.org/.
3. Daly MJ. The computational challenge of linkage analysis: what causes disease? *Computing in Science & Engineering* 1999; **1**(3):18–32.
4. Kruglyak L, Daly MJ, Reeve-Daly MP, Lander ES. Parametric and nonparametric linkage analysis: a unified multipoint approach. *American Journal of Human Genetics* 1996; **58**(6):1347–1363.
5. Kruglyak L, Lander ES. Faster multipoint linkage analysis using Fourier transforms. *Journal of Computational Biology* 1998; **5**(1):1–7.
6. Etnus Inc. *TotalView Multiprocess Debugger User's Guide Version 3.9.0*, May 1999. http://www.etnus.com/.
7. Kuck & Associates Inc. *Guide and Assure Reference Manual (C Edition) Version 3.7*, 1999. http://www.kai.com/.