Performance Programming: Theory, Practice and Case Studies

Module III:
Optimizing Parallel Programs

Outline

- Process and Thread Parallel Models
- Multithreaded Programming Models
- Multithreading Support in UNIX OS's
- Tuning Multithreaded Programs
  - True and False Data Sharing
  - Synchronization and Locking
  - Thread Creation, Stacksize Issues
- Support for Compiler Directed Multithreading
- Parallel Program Analysis Tools
- Message Passing Programming
- Summary

Parallelism Using Processes & Threads

- Multiple Processes or Heavyweight Process model
  - Relies on the traditional process model used by the UNIX OS
  - Any interprocess communication techniques supported in the OS (shared memory, sockets, file input/output, memory map)
  - Not limited by process memory size
  - Higher overhead associated with process creation and destruction
- Multiple Threads or Lightweight Process (LWP) model
  - This model is based on the concept of a thread, which is defined as an independent flow of control within the program with its own context: stack and a set of registers
  - Conserves system resources as the threads share process data and opened files; Lower overhead since thread creation and destruction can be substantially faster
  - Restricted to the shared address space abstraction
  - Large number of threads can lead to high lock contention
- Hybrid Models (eg. MPI+OpenMP) possible

Parallelization Models

- Multithreading Models
  - Compiler Auto-Parallelization
  - OpenMP Compiler Directives
  - Explicit Multithreading Using P-threads
- Multi-Processing Models
  - UNIX fork/exec Model
  - MPI Message-Passing Model
- Hybrid Models
  - MPI + OpenMP or MPI + P-threads

OpenMP Compiler Directives

- Pragmas can be used to instruct the compiler what parts of the program should be parallelized

```
...
!$omp parallel &
!$omp private(i,x), firstprivate(h,n), shared(sum)
!$omp do reduction(+:sum)
do i=1,n ! add points x=(i-0.5)*h x = (i-0.5d0)*h
   sum = sum + 1.0d0/(1.0+x*x)
enddo
!$omp end do
!$omp end parallel
...
```

The program can be compiled on Solaris as

```
example% f90 -fast -openmp example.f90 -o example
```

Similar compiler options available on other systems (discussed later)

- OMP_NUM_THREADS setting controls number of threads

```
example% setenv OMP_NUM_THREADS 2
```

Explicit Multithreading Using P-threads

- Typically more complex than OpenMP
- Irregular, dynamic applications implemented efficiently; allows finer user control on performance.

```
...
for (i=0;i<thr_count;i++) {
    pthread_create(&id_vec[i], &pt_attr[i],
                   thr_sub, (void *)&param_arr[i]);
}
for (i=0;i<thr_count;i++)
    pthread_join(id_vec[i], NULL);
for (i=0;i<thr_count;i++)
sum += param_arr[i].sumloc; picomp = 4.0*sum*h;
...
void *thr_sub(void *arg)
...
{
   for (i=ist;i<=ien;i++) {
     x = ((double)i+0.5)*h; sum += 1.0/(1.0+x*x);
   }
}
```

OpenMP and P-threads

- Parallelization of an existing program
  - OpenMP: can be applied incrementally
  - In P-threads approach, typically higher effort
- Thread-safety and data scoping
  - OpenMP: thread-safety easy as variables can be scoped (shared vs. private)
  - P-threads: need to explicitly privatize variables
- Performance
  - OpenMP: many performance enhancing features, such as atomic, barrier and flush synchronization primitives. Different loop scheduling schemes also supported
  - P-threads: Specialized synchronization primitives need to be build

OpenMP and P-threads (contd.)

- Irregular applications
  - Currently, no support (in OpenMP specification) for dynamically spawning/joining tasks.
  - P-threads APIs are more suitable for this
- Exception and signal handling
  - P-threads have better support for exception and signal handling
  - OpenMP has no support for unstructured control flow/jumps out of parallel constructs. Exception handling not supported directly in the standard
- Conclusion
  - OpenMP and P-threads approaches are suited to different classes of programs

Explicit Multithreading

- Explicit multi-threading applicable to wide range of applications
  - Compute-intensive (CPU-bound)
  - Client-Server, GUI, Signal-handlers...
- Present discussion focuses on
  - Compute-intensive applications
  - Issues specific to UNIX platforms

Multithreading Models

- Performance issues
  - Problem decomposition and granularity
  - Load balancing
  - Synchronization overhead
  - Data sharing overhead
- Multi-threading Models
  - Master-Slave model
  - Worker-Crew model
  - Pipeline model

Master-Slave Model

- Master thread

```
initialization;
create N slave threads;

while (work to do) {

create work for slaves;
call barrier function;

slaves running,
master performs its
processing, if any;

call barrier and wait
for all slaves to arrive;

process results and check
if more work to do;
}

join slave threads;
finalize and perform
any other calculation
```

- Slave thread

```
while (work to do) {

call barrier function
and wait for master to create
work;

process its portion of work;

put results into global
memory or passed parameters;

call barrier and wait till
all slaves finish;
}
call thread exit;
```

Master Slave Model (contd.)

- Only 2 barrier synchronizations required (if slaves are all independent of each other)
- Simplest form: work known in advance and all slaves perform equal chunks. In this case good load balance also
- Often used in compiler parallelization libraries (e.g. Solaris/SPARC OpenMP compiler)

Worker-Crew Model

```
                      Main

Initialize task-queue and create worker threads;
wait for threads to exit;
```

```
                     Worker

for (;;) {
  t = get task from queue;
  if global-->done flag set then break;
  check if max task queue length reached or
  work finished;
  if true then set global->done flag
  (under protection of mutex lock);
  add new tasks to workpile if any (put task);
  check if t is valid task and perform the task;
}
```

Worker Crew Model (contd.)

- Can have *static workpile* or *dynamic workpile* (tasks created at runtime)
- Achieves good load balancein irregular computations but has higher lock contention and data sharing over-head
- Example usage: sparse matrix computations, search al-gorithms, divide and- conquer, loop parallelization [`schedule(dynamic)` clause in OpenMP]

Pipeline Model

Pipeline

```
if stage = 0 { put work in queue 0; /* 0 <= stage <= NSTAGES */ }
for (;;) {
  if (stage > 0) {
      get work from queue (stage-1); /* wait if no work */
      if (task = EXIT or DONE) {
           if (stage < NSTAGES) put EXIT or DONE in queue (stage+1);
           break and then terminate;
      } /* end of if (task = EXIT or DONE) */
  } /* end of if (stage > 0) */
  Perform the work and update shared data;
  if (stage = 0) {
      check if all work done;
      if all work done {put DONE in queue 0; break and terminate;}
      if an error {put EXIT in queue 0; break and terminate;}
      put work in queue 0; }
  else if (stage > 0 and stage < NSTAGES) {
      put work in queue (stage + 1); /* wait if queue being
                                        accessed */
  } /* end of if (stage = 0) */
}
```

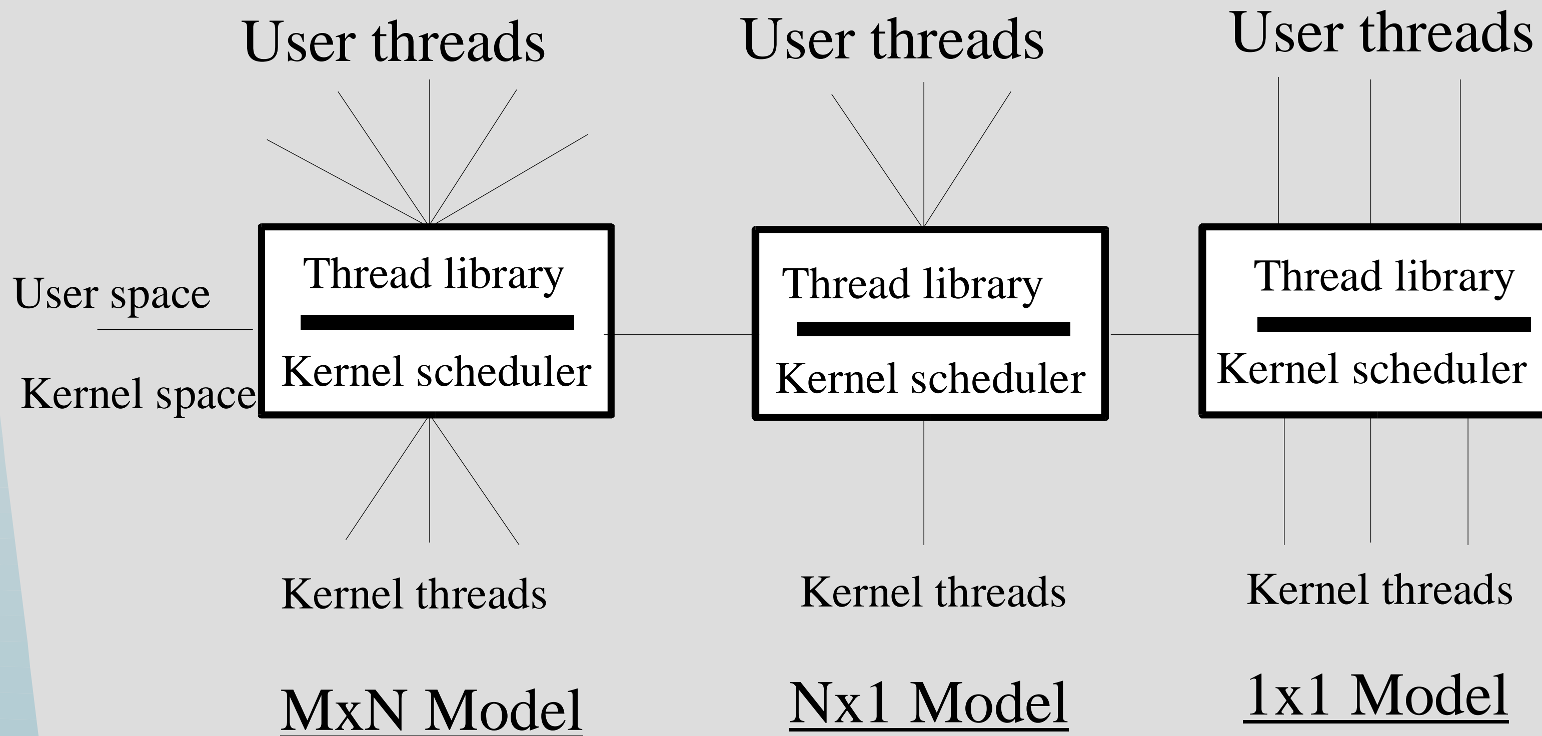
Multithreading via P-Threads

- Historically, vendor specific multithreading libraries (e.g. UI threads on Solaris)
  - Implementations differed making porting hard
- For UNIX systems, IEEE POSIX 1003.1c standard (1995) was developed
  - Implementation of this standard called P-threads or Posix threads
  - Adds following features to enable parallel programming
    - Thread management functions
    - Synchronization functions
    - Thread Scheduling functions (including realtime)
    - Thread-specific data functions
    - Thread cancellation functions
  - Nearly All UNIX systems offer it (Solaris, AIX, HPUX, IRIX, Linux etc.) in addition to proprietary thread libraries

Multithreading via P-Threads (contd.)

- Libraries
  - `libpthread` (on most systems)
  - `libposix4` (on Solaris for Posix Semaphores)
  - `libc` (on Solaris has empty stubs to facilitate program development and debugging)

Thread Scheduling Models



Thread Scheduling Models (contd.)

- MxN and Nx1 models
  - Commonly referred to as unbound threads (process scope); can migrate within the pool of kernel threads in the process
  - Nx1 is just a special case of MxN model and an intermediate step in thread library evolution
- 1x1 model
  - Also referred to as bound threads (system scope); user thread sticks to a particular kernel thread for its lifetime
- Traditionally, thread creation and destruction faster in MxN and Nx1 models compared to 1x1 model
- However, recent improvements in OS kernel make the speed difference virtually insignificant
  - Solaris 9 (for example) has switched to 1x1 model

Building Threaded Programs

- POSIX APIs developed for C programs (no direct support for Fortran)
- POSIX function names: `pthread_`
- For POSIX thread programs include: `pthread.h`

```
#include <stdio.h>
#include <pthread.h>

void *thr_foo(void *);

pthread_attr_t t_attr;
pthread_t id;
int param=1;

pthread_attr_init(&t_attr);
pthread_attr_setscope(&t_attr, PTHREAD_SCOPE_SYSTEM);
pthread_attr_setstacksize(&t_attr, 1000);
pthread_create(&id, &t_attr, thr_foo, (void *)&param);
pthread_join(id, NULL);
```

Building Threaded Programs

- Usually several ways to compile and link P-thread programs For example on Solaris any of following will work:

```
example% cc -D_POSIX_C_SOURCE=199506L
         example_thread.c -lpthread \
         -o example_thread

example% cc -D_REENTRANT example_thread.c
         -lthread -o example_thread

example% cc -mt example_thread.c -o example_thread
```

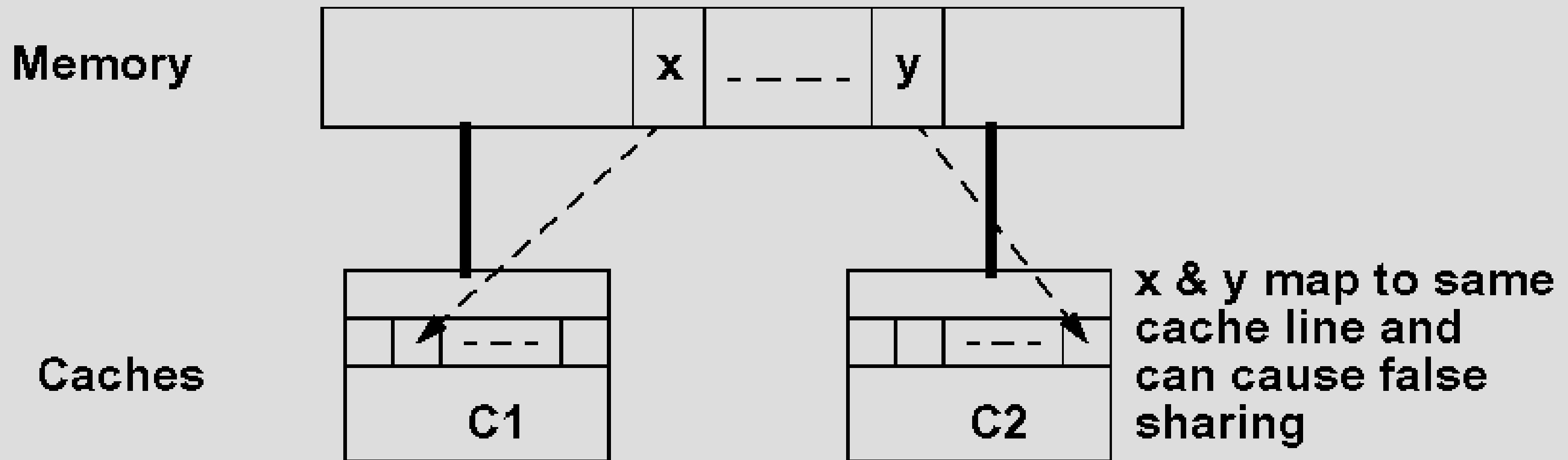
On Linux, following can be used:

```
example% gcc example_thread.c -o example_thread \
         -lpthread
```

Data Sharing

- **Data sharing**
  - True sharing: inevitable in multi-threaded programs
  - False sharing: consequence of cache-line granularity based cache coherence
- **True sharing**
  - Complete elimination rarely possible
  - Leads to synchronization as protected access to shared data required
  - Restructure program or change algorithm to decrease
- **False sharing**
  - Leads to performance degradation
  - Can be completely eliminated in many cases with program restructuring, data structure changes, loop scheduling modifications

False Data Sharing



Multiple copies of the same cache line may exist in the MP system

Cache coherency ensures correct cache line (hence data) is used

Shared lines may cause <u>False</u> sharing of data and performance loss
- **Increase in number of cache misses**
- **Increased miss penalty due to contention for cache line ownership**

Eliminating False sharing

- Restructure program loops and constructs
- Changing data structures
- Data duplication
- Changing loop scheduling parameters

```
double array[SIZE][4], sum, s[4];
...
s[mynum] = 0.0;
for(iter1=0;iter1<MAXITER;iter1++){
  for (j1=0;j1<SIZE;j1++){
     s[mynum] += array[j1][mynum];
               /* False Sharing Occurs */
  }
}
```

False Sharing (contd.)

- Compiling and running on a Sun Enterprise 4500

```
example% cc -mt example_false_sharing.c -o
example_false_sharing
example% example_false_sharing
sum = 20838169.391802
RUNTIME (One thread) : 3.2394 Seconds
sum = 20838169.391788
RUNTIME (Four threads) : 8.8611 Seconds
```

- What happens?
  - Cache line containing array `s` repeatedly invalidated causing increase in memory traffic
  - Measure cache snooping (can be done on Solaris systems using `cputrack` or `cpustat` tools)

False Sharing (contd.)

Examining `EC_snoop_inv` and `EC_snoop_cb` events

```
0.045    4326   3   tick   18 16
2.093    4326   1   tick   100 107
2.014    4326   2   tick   0 0
0.045    4326   3   tick   0 0
...
3.474    4326   8   tick   0 0
3.457    4326   1   tick   0 0
6.014    4326   2   tick   0 0
3.474    4326   3   tick   0 0
6.093    4326   4   tick   1023090 1019874
6.093    4326   5   tick   1023104 1019008
6.094    4326   6   tick   1023361 1020391
6.094    4326   7   tick   1023328 1019657
3.474    4326   8   tick   0 0
3.457    4326   1   tick   0 0
7.014    4326   2   tick   0 0
3.474    4326   3   tick   0 0
7.104    4326   4   tick   1024083 1021300
7.104    4326   5   tick   1023743 1019784
7.104    4326   6   tick   1024498 1021366
7.103    4326   7   tick   1024398 1020483
```

False Sharing (contd.)

- Padding Array s

  double array[SIZE][4], sum, s[4][8];

```
...
1.071 4380 1 tick 320 339
1.011 4380 2 tick 4   3
0.048 4380 3 tick 19  15
2.091 4380 1 tick 194 206
2.011 4380 2 tick 0   0
0.048 4380 3 tick 0   0
...
3.483 4380 1 tick 22  187
4.001 4380 2 tick 0   0
3.501 4380 3 tick 18  250
4.091 4380 4 tick 260 19788
4.091 4380 5 tick 381 2167
4.091 4380 6 tick 275 295
4.091 4380 7 tick 112 162
3.501 4380 8 tick 12  2
```

False Sharing (contd.)

false sharing

1 CPU

4 CPUs

no sharing

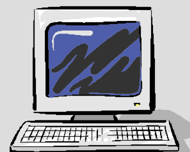
1 CPU

4 CPUs

0 1 2 3 4 5 6 7 8 9 10 11

Runitme in seconds

Times on a Sun Enterprise 4500 system (Solaris 8)



False Sharing (contd.)

- False Sharing in Mutex Locks
  - Avoid static vector of mutexes

```
typedef struct {
pthread_mutex_t sync_mutex;
#ifdef PAD
char pad[PADSZ];
#endif
} mutexlock;
mutexlock syncarr[MAXTHREAD];
```

- PAD_SZ depends on E-cache line/block size and size of structure `pthread_mutex_t; some` L2-cache line sizes

  UltraSPARC-II/III: 64-bytes
  Power-4: 128-bytes
  Itanium-2: 128-bytes
  Intel Xeon: 32-bytes
  Alpha 21264: 64-bytes

Synchronization & Locking

- Threads communicate via shared data
  - Communication is very efficient due to shared address space
  - Shared data access may require synchronization
- POSIX threads provide many Synchronization functions
  - Mutex locks, semaphores, condition vars, thread join
  - Primitives such as barrier can be built using these
- Synchronization overhead
  - Latency of synchronization (cost of executing synchronization function)
  - Waits induced by synchronization (contention to access shared resource)
- To decrease synchronization overhead important to under-stand how they are implemented and interact with underlying hardware

Synchronization & Locking (contd.)

- Hardware considerations
  - Atomic memory operations
  - Global data visibility & memory consistency model
- Atomic memory operations
  - An operation that is indivisible. Once started is guaranteed to finish without any interruptions
  - Performed at level of machine instruction but can also correspond to execution of a statement in a high level language.

Synchronization & Locking (contd.)

- Global data visibility: effects of instruction reordering and load (store) buffers as specified by the memory consistency model implemented in processor architecture
  - Memory model: A set of rules that constrains order of completion of loads and stores with respect to each other
  - Cache coherency relates to consistency of same memory locations in different caches
  - Memory consistency relates to consistent ordering of memory operations to different location.

Synchronization & Locking (contd.)

- Atomic Instructions
  - SPARC V9
    - Atomic Test and Set (`ldstub`)
    - Compare and Swap (`cas`) : SPARC V9 (more powerful, can be used to implement wait-free synchronization)
    - Atomic Exchange (`swap`)
  - IA-64 (Semaphore instructions)
    - Exchange/Compare and Exchange (`cmpxchg`)
    - Fetch and Add – Immediate (`fetchadd`)

Synchronization & Locking (contd.)

- Memory Ordering Instructions
  - SPARC V9
    - Store Barrier (`stbar`)
    - Memory Barrier (`membar`): can to used to order loads and stores with respect to each other
    - Current UltraSPARC/Solaris systems implement Total Store Order (TSO) memory model
  - IA-64
    - Sync/Serialize (`sync, srlz`)
    - Fence (`mf, inva, mwb`)

Synchronization & Locking (contd.)

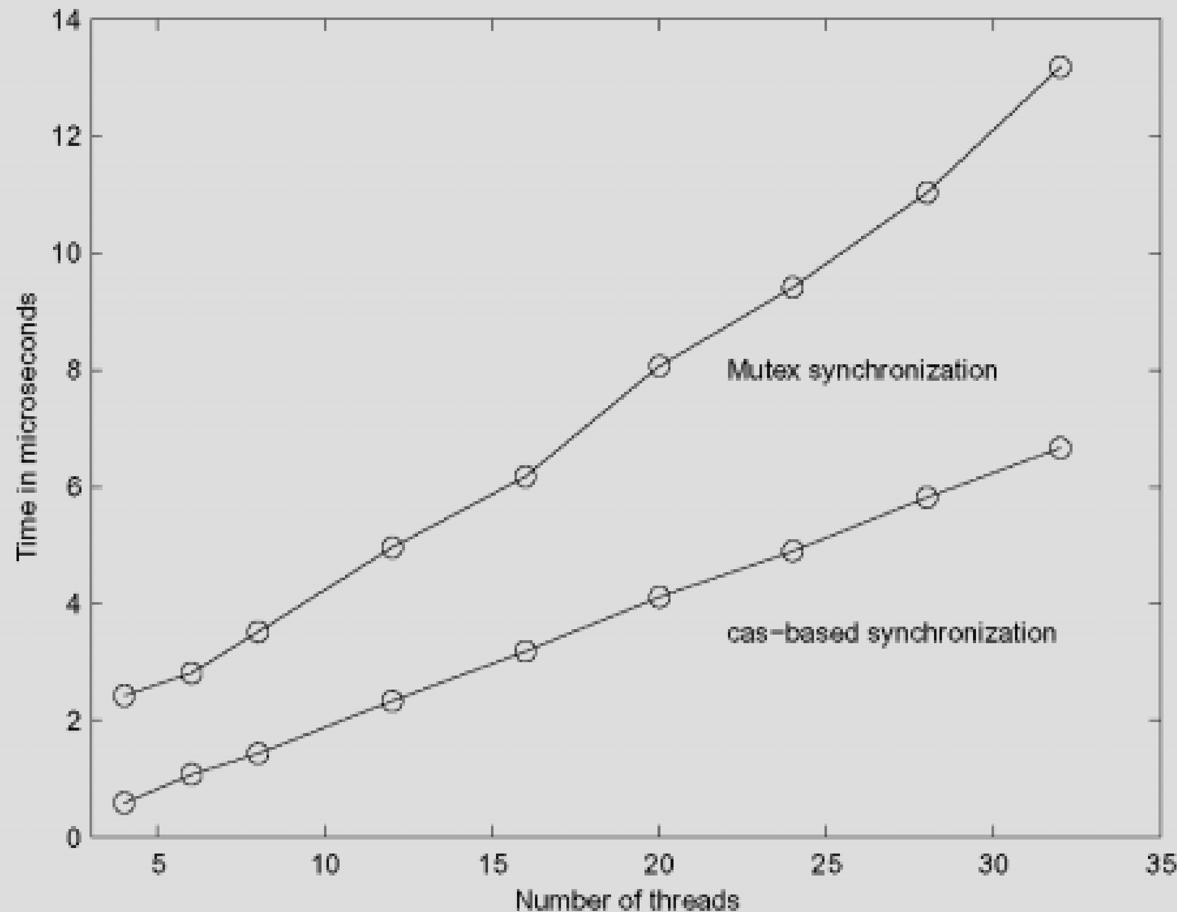
- Pseudo code for Mutex lock (using SPARC `ldstub` instruction)

```
Lock_function
membar #StoreLoad  ! assume TSO model

                    ! load 1 byte from location

                    ! 'adress' & store 1's into it.
retry: ldstub address, register
cmp register, 0     ! check if loaded value is 0
beq gotit           ! if value is 0, we got the lock
call delay          ! if value is 1, then lock is held
                    ! by someone else. Go to sleep
                    ! before retrying
jmp retry           ! Wake up and rety to acquire lock
gotit: return

Unlock_function
membar #StoreStore ! Ensure all prior stores completed
stub 0, address     ! Store 0 in 'address' byte
```

Synchronization & Locking (contd.)

Locking Under contention: cas vs. mutex_lock for fetch_and_add

`counter = counter + 1 ;` **/(fetch & add primitive) */**



Times on E10000, 64-cpus, 400MHz US-II, Solaris 8

Synchronization & Locking (contd.)

- Contention for shared lock scales faster than linear with number of threads
- Avoid small and frequent critical sections
  - Merge them into larger critical sections
  - Consider using fast atomic synchronization wherever applicable
  - Difficult to prescribe a recipe for optimum size of critical section
- Avoid use of "Self-Synchronization"

```
/* THE FOLLOWING SHOULD NEVER BE DONE */
volatile int mylock = 0;
while (mylock != 0) ;
mylock = 1;
......execute critical region code.......
mylock = 0;
```

Thread Creation Issues

- Common questions related to thread creation and management
  - How many threads to create?
  - Whether to use bound or unbound threads?
  - Whether to create threads statically or dynamically?
  - Whether to use pool of threads?
- For compute intensive applications
  - Number of threads less than number of processors on system
  - Use bound threads
  - If amount of work known in advance use static thread creation otherwise use dynamic thread creation (i.e. create threads as needed)
  - If threads perform small amount of work, then overhead of creation/joining could be high and pool of threads should be considered. Used often in Parallelizing Compilers.

Thread Creation Issues (contd.)

- Example extracted from a discrete event simulation
  - small amount of work (extracted from event queues)
  - thread create/join
  - thread pool using Semaphores
- Thread create/join

```
real 1:06.560
user 47.560
Sys 1:13.837
```

- Thread pool using Semaphores

```
real 44.471
user 31.086
sys 1:30.778
```

(Time on 12-cpu Enterprise 4500, Solaris 8)

Thread Stack Size

- Changing thread stack size
  - Many threads created/destroyed in the program: smaller stacksize if allowed leads to more efficient memory usage.
  - Requirement of system libraries (most require at least 8 KB stack size) and function call depth.
  - Increasing thread stack size: impact on virtual memory usage of process and overhead

- Default Stack Size
  - Varies (on Solaris 9: 1 MB for 32-bit, 2 MB for 64-bit)

Thread stack size (contd.)

32 MB
16 MB
12 MB
8 MB
4 MB
2 MB
1 MB

0 1 2 3 4 5 6 7
Runtime in seconds

30 threads on 12-cpu E4500 (400MHz US-II, Solaris 8)

Effect of stacksize on system scope thread creation/join

Compiler Parallelization

- Automatic Parallelization
  - Based on automatic dependence and alias analysis compiler re-structures the program for parallel processing with no user inter-vention
  - Can parallelize “well-behaved” loops effectively but not very flexible in handling more complex cases
- Directive-based Parallelization
  - Proprietary vendor-specific directives
  - OpenMP directives (Industry standard, portable)
  - Directives can substantially enhance applicability and efficiency of compiler parallelization model

Compiler Parallelization (contd.)

- Porting from vendor-specific directives to OpenMP
  - Vendor directives usually have a one-to-one mapping to OpenMP (most vendors provide porting documentation; E.g. Sun ONE Studio Compiler Collection 7 OpenMP API User's Guide)
  - Some things to watch for
    - All variables need to be explicitly scoped in OpenMP (vendor directives may assume different defaults for shared and private variables)
    - Since some vendor directives do not have one-to-one mapping to OpenMP directives, unpredictable effects may occur if OpenMP and vendor-specific directives are mixed in the same program
    - OpenMP provides a powerful and rich parallelization model. It might be possible to get better performance by exploiting OpenMP features not available in vendor directive set

Compiler Parallelization (contd.)

- Porting from vendor-specific directives to OpenMP (an example)

Sun Fortran Directive	*Equivalent OpenMP Directive*
C$ PAR DOALL [qualifiers]	!$omp parallel do [qualifiers]
c$par doserial	No exact equivalent, Can use
	!$omp master ... loop... !$omp end master
c$par taskcommon block[,...]	!$omp threadprivate (/block/[,...])
SCHEDTYPE(FACTORING(m))	No OpenMP Equivalent
SCHEDTYPE(GSS(m))	schedule(guided, m) default m is 1

SGI Fortran Directive	*Equivalent OpenMP Directive*
c$doacross	!$omp parallel do
c$par parallel	!$omp parallel
c$par pdo	!$omp do

Compiler Parallelization Library

Master thread

Pool of slave threads created

Serial Portion (slaves idle)

Program Timeline

Parallel Construct 1

Last Parallel Construct

Slave threads join master; it terminates program

Compiler Parallelization Library (contd.)

- This approach is used in many compilers (eg. Sun compiler, KAI guide library)
- Pool of threads using master-slave model
- Slave threads are created once, when the first parallel region is encountered, and then used in all parallel regions of the program.
- The body of the parallel construct is then extracted and placed in a separate subroutine called an outlined function.
- The compiler then inserts a call to a driver routine executed by the master thread in place of the original parallel region

Compiler Parallelization (contd.)

- Automatic parallelization options in many compilers for Fortran (77, 90/95), C and C++ programs
  - Sun compilers: `-xautopar, -xparallel`
  - HP compilers: `+Oparallel, +Oautopar`
  - Compaq compilers: `-hpf`
  - SGI MIPSpro compilers: `-apo` (or `-pca`, `-pfa`)

- Some caveats about automatic parallelization
  - Usually outermost loop in a loop-nest auto-parallelized
  - Usually additional options required (eg. For Sun compilers `-xO3` required and `-xdepend, -xrestrict, -xalias_level` help)
  - Parallelization usually disabled/suppressed in following cases:
    - Function call or I/O in loop
    - Conditional exit from loop (`goto` statement)
    - Loop iterations change the variable aliased through a pointer or an EQUIVALENCE statement.
  - Loops whose iterations update the same scalar variable do not get parallelized. Exception: reduction (a computation that transforms a vector into a scalar); E.g. For Sun compilers option `-xreduction` required.

OpenMP Compilation Options

Compiler	Fortran	C/C++
Sun ONE SCC7 (Solaris/SPARC)	-openmp	-xopenmp
HP Compiler (HPUX11i/Itanium-2)	+Oopenmp	+Oopenmp
HP Compiler (HPUX11i/PA-RISC)	+Oopenmp	+Oopenmp
Compaq Compiler (TRU64/Alpha 21264)	-omp	-omp
SGI MIPSpro 7.3 (IRIX/MIPS R14000)	-mp	-mp

Runtime Settings

- Typically, environment variables to control number of threads, thread stacksize, spin-wait behavior of threads, thread placement etc.
- Number of threads
  - `OMP_NUM_THREADS` envar used (part of OpenMP standard)
  - `OMP_DYNAMIC` (to enable/disable dynamic thread adjustment)
- Stacksize of worker/slave threads
  - Sun compiler: `STACKSIZE` envar

    `setenv STACKSIZE 8192` (8MB stacksize)

    Default is 2MB (32-bit) and 4MB (64-bit)
  - HP compiler (HPUX/PA-RISC): `CPS_STACK_SIZE`

Runtime Settings

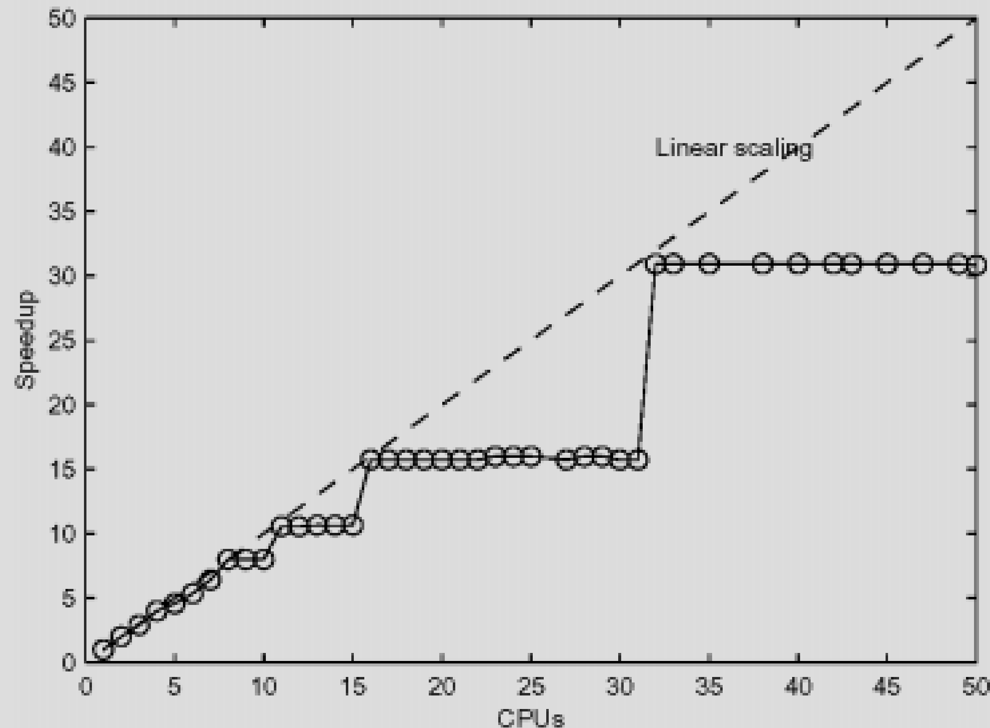
- Spin wait timeout: controls behavior of threads in idle portions of the program
  - Sun compiler: by default threads spin-wait. To set the wait time to 10ms

    ```
    setenv SUNW_MP_THR_IDLE 10ms
    ```
  - HP compiler: `MP_IDLE_THREAD_WAIT` envar can be used to achieve similar effect
  - Compaq compiler: `MP_SPIN_COUNT` and `MP_YIELD_COUNT` envars can be used for similar effect
- Thread placement: can have big impact on machines with non-uniform memory access features
  - Sun Solaris: `MT_BIND_PROCESSOR` envar
  - SGI IRIX: `_DSM_PLACEMENT` envar

Stair Stepping Effect

- Parallel speedup can be limited (and non-linear) if number of iterations in a parallelized loop is low compared to number of threads

```
c$par doall private(j) reduction(s)
do j=1,32
  call sleep(10)
  s=s+1
enddo
```

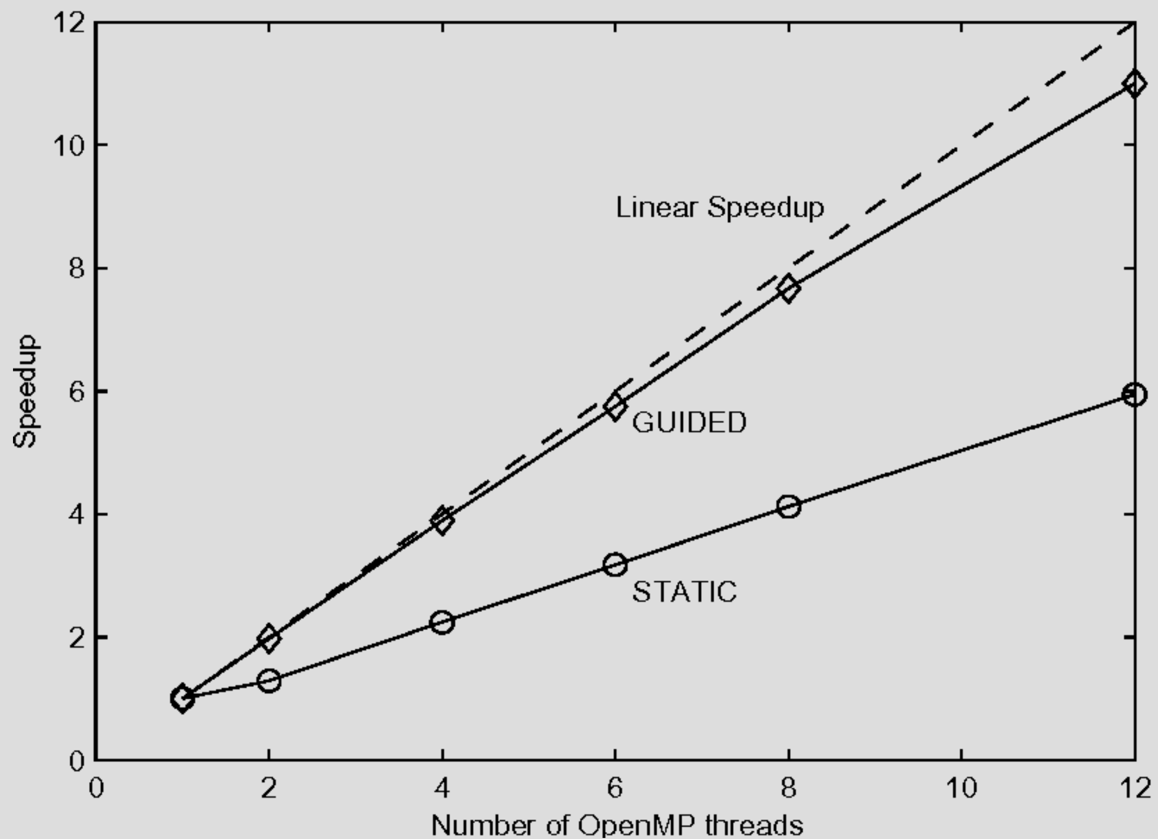


64-cpu Sun E1000 system (Solaris 8)

OpenMP Runtime Scheduling Type Setting

Example: Triangular by Square Matrix Multiplication

```
!$omp parallel &
!$omp private(i,j,k), shared(a,b,c)
!$omp do schedule(runtime)
do j=1,idim
do i=1,idim
atmp=a(i,j)
do k=1,j
atmp = atmp+b(i,k)*c(k,j)
enddo
a(i,j)=atmp
enddo
enddo
!$omp end do
!$omp end parallel
```



OpenMP Synchronization Issues

- The OpenMP specification includes `critical`, `atomic`, `flush`, and `barrier` directives for synchronization purposes
- Additionally, there are functions for user-inserted locks, in a manner similar to mutex locks in P-threads library
- When using critical directive, it is recommended to associate it with a name since all unnamed critical sections map to the same name
- OpenMP flush directive guarantees the atomicity and memory consistency. In order to avoid false sharing, the variables that are flushed should preferably be placed on different cache lines by padding or not declaring them contiguously

OpenMP Synchronization Issues (contd.)

- OpenMP `ordered` directive might be useful in a routine where threads perform I/O or other library calls requiring sequential order, but its use in a parallelizable do loop should be avoided as much as possible
- Wherever the logic of the program allows, the `nowait` qualifier should be used to eliminate the implicit barrier at the end of the parallel `do` loop that is required by the standard

Memory Bandwidth Requirement

- As more threads are added in the parallel program, additional simultaneous requests for memory accesses (reads or writes) are generated, increasing the traffic on the system interconnect
- Overall speedup depends on the system being able to satisfy the independent memory requests generated by the different processors; A parallel application's bandwidth requirement might exceed what the hardware can deliver and the application will no longer scale, even if more processors are used to run it.

Memory Bandwidth Requirement (contd.)

Number of CPUs	Measured STREAM Copy B/W (GB/s)	Total STREAM Copy B/W (GB/s)
1	0.29	0.44
2	0.57	0.85
4	0.99	1.49
6	1.25	1.88
8	1.44	2.16
10	1.53	2.3
12	1.53	2.3

Results on an Enterprise 4500 system (400MHz US-II, Solaris 8)

Analysis Tools for OpenMP Programs

- Difficult to create tools for parallel programs but now many effective tools (from different vendors) exist
- Correctness checking tools
  - OpenMP programs have difficult correctness problems: variable scoping, data storage conflicts, work distribution, data races and synchronization conflicts
  - Some good tools: Intel-KAI Assure (runtime checks), Forge Explorer (from now defunct APR), static verification(`-XlistMP`) capability in Sun Fortran-90 compiler
- Performance analysis tools
  - PC-Sampling based (statistical) tools such as SGI perfex, Sun Performance Analyzer and others
  - Instrumentation based tools (such as gprof, prof). These should be avoided as intrusive and inaccurate results

Example Sun Performance Analyzer (1 of 3)

- Tool can be used to obtain profile data for OpenMP and multithreaded programs (no recompilation/relinking required)

Performance Analyzer [test.1.er]

File View Timeline Help

O A D M P N Find Text:

PCs | Timeline | LeakList | Statistics | Experiments

Functions | Callers-Callees | Source | Lines | Disassembly

User CPU (sec.)	User CPU (sec.)	Wall (sec.)	Wall (sec.)	Name
7.835	7.835	2.061	2.061	<Total>
3.843	7.675	0.290	1.921	_$d1A27.factk_full_
2.932	2.932	1.631	1.631	__mt_EndOfTask_Barrier_
0.901	0.901	0.	0.	__mt_flush_
0.120	0.120	0.	0.	__mt_WaitForWork_
0.030	0.030	0.030	0.030	backs_full_
0.010	0.010	0.010	0.010	normk_full_
0.	1.961	0.	2.001	MAIN_
0.	7.675	0.	1.921	_$p1B20.factk_full_
0.	0.	0.	0.060	__f90_init
0.	0.	0.	0.060	__f90_initio_a
0.	1.921	0.	1.921	__mt_MasterFunction_
0.	5.874	0.	0.	__mt_SlaveFunction_
0.	4.743	0.	0.290	__mt_WorkSharing_
0.	4.743	0.	0.290	__mt_runLoop_int_
0.	7.675	0.	1.921	__mt_run_my_job_
0.	1.961	0.	2.061	_start
0.	5.874	0.	0.	_thread_start
0.	1.921	0.	1.921	factk_full_
0.	1.961	0.	2.061	main
0.	0.	0.060	0.060	malloc
0.	0.	0.020	0.020	readk_full_
0.	0.	0.020	0.020	savek_full_

Summary | Event | Legend

Data for Selected Object:

Name:	_$d1A27.factk_full_
PC Address:	2:0x00004DA0
Size:	872
Source File:	factk_full.f
Object File:	(unknown)
Load Object:	<vf>
Mangled Name:	
Aliases:	

Process Times (sec.) / Counts

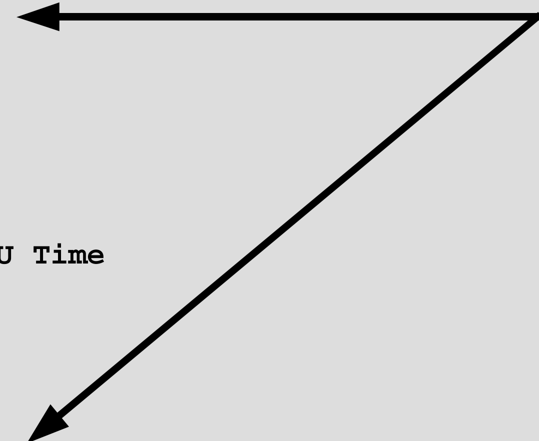
	Exclusive	Inclusive
User CPU:	3.843 (49.0%)	7.675 (98.0%)
Wall:	0.290 (14.1%)	1.921 (93.2%)
Total LWP:	3.853 (48.5%)	7.685 (96.7%)
System CPU:	0.010 (16.7%)	0.010 (16.7%)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0. (0. %)	0. (0. %)

4 thread run on SunFire 68K. Tool shows time spent in OpenMP library functions (__mt_xxx functions). Time in `__mt_EndofTask_Barrier_` indicates load-imbalance

Example Sun Performance Analyzer (2 of 3)

- Can obtain per-thread data to locate load-imbalance (use `er_print` command line version to see the 4-thread run data from previous slide)

```
(er_print) thread_list
Exp Sel Total
=== === =====
  1   1-6     6
(er_print) thread_select 1
Exp Sel Total
=== === =====
  1   1       6
(er_print) functions
Functions sorted by metric: Exclusive User CPU Time
Excl.     Incl.      Name
User CPU  User CPU
 sec.      sec.
1.961     1.961      <Total>
1.631     1.631      __mt_EndOfTask_Barrier_
0.290     1.921      _$d1A27.factk_full_

(er_print) thread_select 4
Exp Sel Total
=== === =====
  1   4       6
(er_print) functions
Functions sorted by metric: Exclusive User CPU Time
Excl.     Incl.      Name
User CPU  User CPU
 sec.      sec.
1.961     1.961      <Total>
0.981     0.981      __mt_EndOfTask_Barrier_
0.831     1.921      _$d1A27.factk_full_
(er_print)
```



Example Sun Performance Analyzer (3 of 3)

- Timeline viewer: shows events recorded in the experiment as a function of time (on a per-thread or per-MPI process basis) alongwith the callstack of the program

Performance Analyzer [test.1.er]

File View Timeline Help

Find Text:

PCs | Timeline | LeakList | Statistics | Experiments

Functions | Callers-Callees | Source | Lines | Disassembly

Summary | Event | Legend

Data for Current Timeline Selection

Experiment Name:	test.1.er
Event Type:	Profiling Data
Leaf Function:	__mt_EndOfTask_
Timestamp (sec.):	0.910959
LWP:	1
Thread:	1
CPU:	(unknown)
Duration (msec.):	10.007
Micro State:	User CPU

Call Stack for Selected Event

__mt_EndOfTask_Barrier_ + 0x00000088
_$d1A27.factk_full_ + 0x00000000
_$p1B20.factk_full_ + 0x00000180
__mt_run_my_job_ + 0x000001D0
__mt_MasterFunction_ + 0x0000038C
factk_full_ + 0x00000290
MAIN_ + 0x00000260
main + 0x00000030
_start + 0x00000108

Example Intel-KAI Guideview Tool (1 of 2)

- Graphical tool to analyzer performance of OpenMP programs

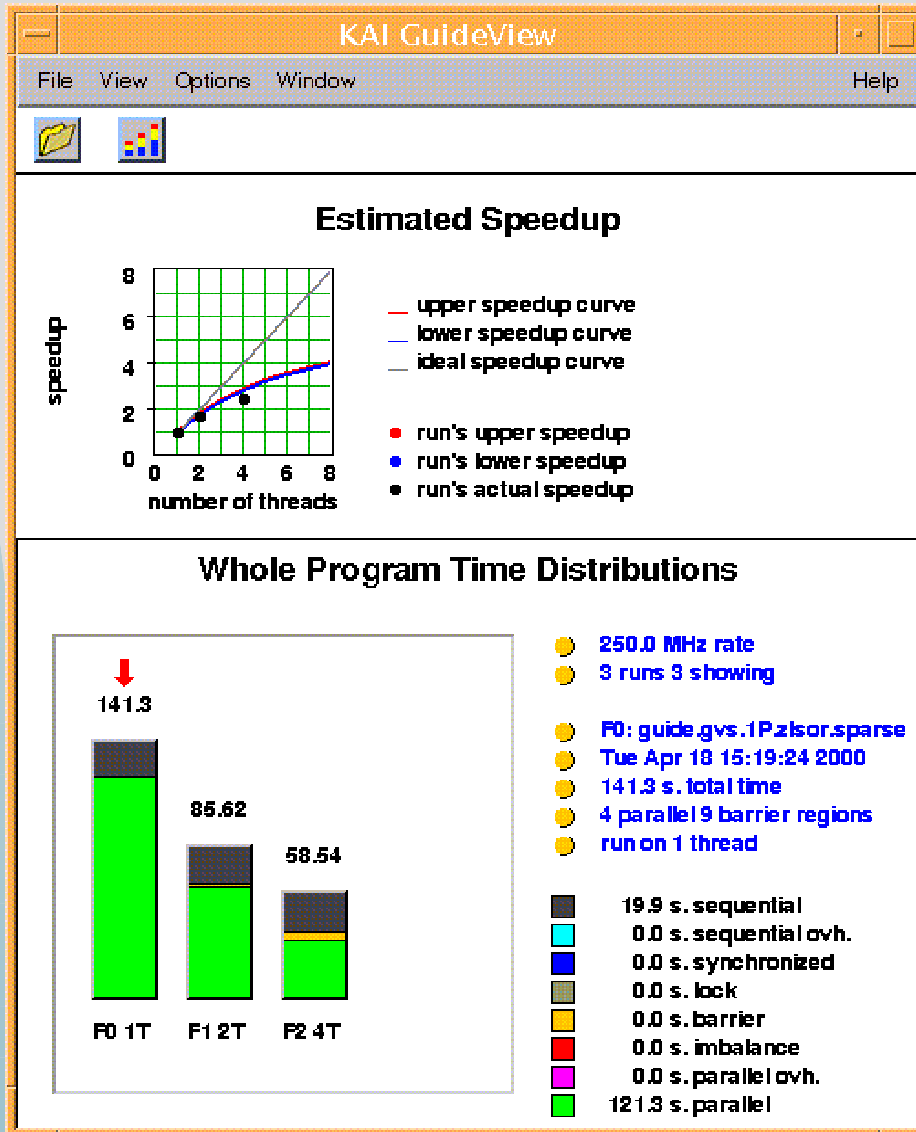
  - Link the openmp program with guidestats library
    `guide{f77,f90,cc} $(FLAGS) *.o -Wgstats`

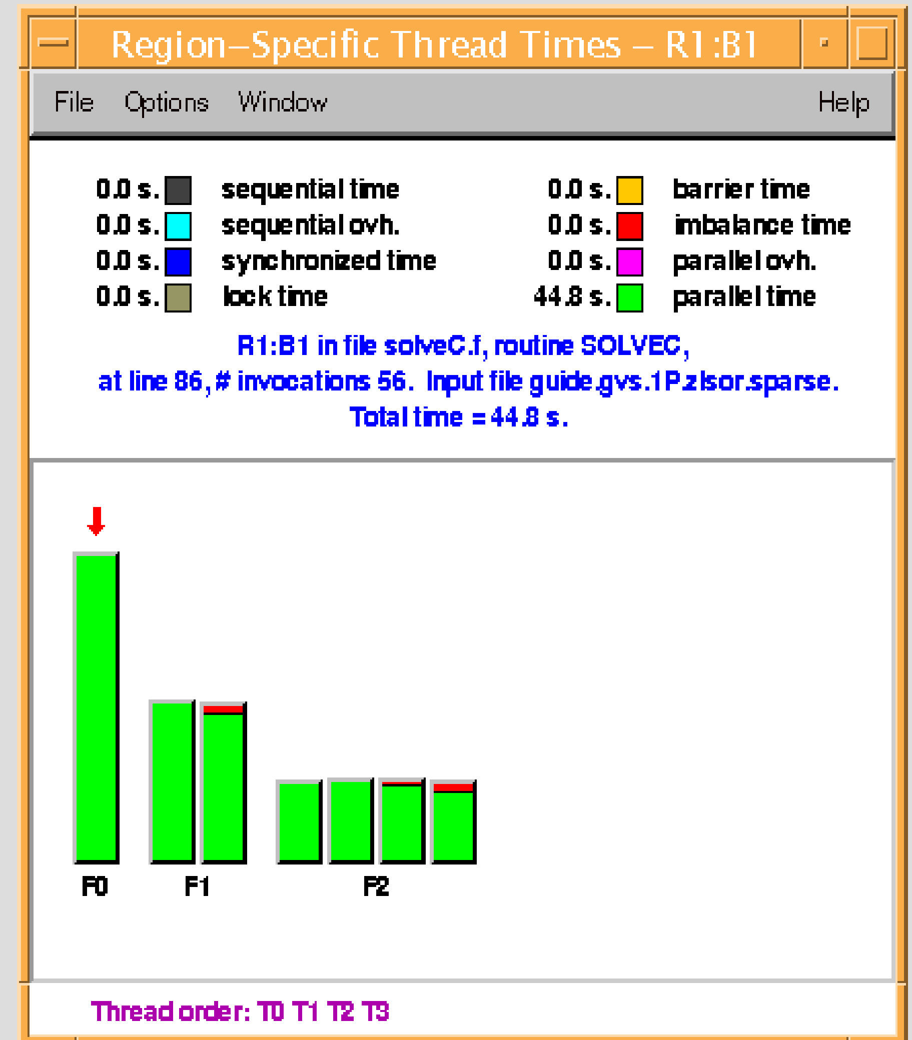
  - Run the program, guide.gvs (ascii file) generated, can be visualized with guideview tool
    `guideview *.gvs`

  - Tool can be used to analyze files from multiple runs
    - Sequential time, sequential overhead, synch. time, lock time, barrier time, imbalance time, parallel overhead and parallel time for each thread can be seen
    - Times can be viewed as parallel region times or thread times
    - Times can be viewed per each parallel region or `omp do` loop on a line by line and function by function basis in the program and can be compared side-by-side for the different runs (1P, 2P, 4P etc.)
    - Times can also be viewed per thread

Example Intel-KAI Guideview Tool (2 of 2)

- Overall times for 1,2,4 threads
- Region specific thread times (for a hot region)





Message Passing Programs

- Performance issues in message passing programs
  - Organization of computing tasks
  - Distribution of workload
  - These two affect: implementation complexity, communication and synchronization requirements, ratio of computation to communication, load balance
- Communication requirements characterized by
  - Volume (size and number of messages)
  - Frequency of message passing (temporal aspect of communication)
  - Communication pattern (nearest neighbor vs. global)
  - Tolerance (ability to overlap communication with computation)
- Organization of Computing tasks: SPMD, MPMD, Hybrid

Message Passing Programs (contd.)

- Two approaches commonly used for workload distribution
  - Functional decomposition: different functions/tasks performed in parallel
  - Data decomposition: parallel tasks perform same computational function but operate on different data portions
- Functional decomposition -
  - May involve spawning new tasks dynamically or attaching to pre-existing but separate tasks. Dynamic process management feature of MPI-2 standard facilitates this feature.
- Data decomposition
  - More common approach in message passing scientific HPC applications
  - Static data decomposition: workload of each task fixed during computation
  - Dynamic data decomposition: workload dynamically varies at runtime - Static data decomposition: more common in message passing scientific HPC applications
  - Static data decomposition equivalent to domain decomposition in many applications such as arising in fluid dynamics, elasticity etc.

Message Passing Programs (contd.)

- Static Data Decomposition
  - Structured mesh based applications
  - Unstructured mesh based applications
- Partitioning in structured mesh/grid based applications
  - Parallel by point
  - Parallel by line
  - Parallel by plane
- Partitioning in unstructured mesh/grid based applications
  - Methods are based on graph theoretic approaches
  - Problem is NP-complete and heuristic methods have been developed
  - RCB: Recursive Coordinate Bisection
  - RGB: Recursive Graph Bisection
  - RSB: Recursive Spectral Bisection
  - MLND: Multi-Level Nested Dissection

MPI

- MPI (Message Passing Interface) has become the *de-facto* standard for message passing programming
  - Provides a library of routines for interprocess communication APIs support Fortran, C and C++. MPI 2.0 is the current standard (`http://www.mpi-forum.org`)
  - Is the only reasonable way to program a cluster
- MPI features
  - Communication: point-to-point, collective, synchronous, blocking, non-blocking, buffered
  - Synchronization: barrier, wait, waitall
  - Parallel I/O
  - One-sided communication
  - Dynamic process management (spawning, connection)

MPI (contd.)

- MPI Implementations
  - Public-domain (eg. MPICH from ANL)
  - Proprietary vendor-optimized (eg. Sun, IBM, HP, SGI others)
  - Recommend using vendor optimized implementations as highly tuned for underlying platform
- We will discuss some features of Sun MPI implementation as an example

Example Sun MPI Implementation

- Highly optimized MPI implementation with support for C, C++, Fortran 77 and Fortran 90 (full MPI 1.1 std. + MPI-2 std. subset support)
  - Thread-safety
  - Support for the Cluster Runtime Environment (CRE), Load Sharing Facility (LSF), Sun Grid Engine (SGE) and other job-scheduling software tools
  - Support for the Prism performance and debugging environment
  - Optimized collective communication for SMP and clusters of SMPs
  - Numerous environment variables for fine-tuning communication performance
  - Support for MPI-2 features (MPI I/O, dynamic process spawn, limited support of one-sided communication)
  - Process yielding and coscheduling to decrease performance degradation in over-subscribed runs.

Example Sun MPI Implementation (contd.)

- Building and running programs using Sun MPI
  - Default location is /opt/SUNWhpc
    - commands in /opt/SUNWhpc/bin
    - libraries in /opt/SUNWhpc/lib
    - include files in /opt/SUNWhpc/include
  - MPI library:
    - 32-bit version in /opt/SUNWhpc/lib/libmpi
    - 64-bit version in sparcv9 sub-dir
    - thread-safe version: libmpi_mt
    - Fortran programs: mpif.h include file
    - C & C++ programs: mpi.h include file
  - Use `mpcc, mpf77, mpf90` and `mpCC` drivers for compilation and linking. These pass right options to the compiler driver and linker

```
example% mpcc -fast example.c -lmpi
example% mpf90 -fast example.c -xarch=v9 -lmpi_mt
```

  - Running MPI executables

```
example% mprun -np 4 sample_mpi_job
example% bsub -n 4 sample_mpi_job
```

Example Sun MPI Implementation (contd.)

- Sun MPI environment variables: many envars to tune message passing programs
  - These envars can be categorized into six general areas: informational, general performance tuning, point-to-point performance tuning, numerics, tuning rendezvous message exchange protocol, miscellaneous
- Some recommended settings

```
example% setenv MPI_PRINTENV 1        #print MPI envar values
example% setenv MPI_SHOW_ERRORS 1     #Show errors in MPI lib
example% setenv MPI_SPIN 1            #Spin during waits in MPI calls
example% setenv MPI_PROCBIND 1        #Bind MPI process to cpu
example% setenv MPI_SHM_CYCLESTART 0x7fffffff
                                       #Suppress cyclic msg. Passing
example% setenv MPI_POLLALL 0          #Suppress polling for sys. buffers
                                       #Increase msg buffer size
example% setenv MPI_SHM_SBPOOLSIZE 20000000
example% setenv MPI_SHM_NUMPOSTBOX 256
example% setenv MPI_EAGERONLY 0        #use rendezvous protocol
```

Summary

- Choice of parallel model important and should be done at initial stages
- UNIX OS's provides extensive MT support (multiple thread library implementations). Performance issues to consider
  - Data sharing overhead: true sharing inevitable, false sharing can be eliminated
  - Synchronization overhead: latency of synchronization and impact of contention vs. size of critical section.
  - Use spin locks and fast atomics based primitives where applicable
  - Thread creation: pool of threads over thread create/join
  - Thread stack size: impact on program virtual memory usage and overhead in thread creation/management

Summary (contd.)

- Extensive compiler parallelization support (OpenMP for Fortran and C)
  - loop scheduling type, synchronization and variable scoping are important for performance in OpenMP programs
- Performance analysis tools can be used to generate performance analysis and profile data of multi-threaded applications
- Extensive support for MPI programming
  - Use optimized vendor MPI implementations