Coding for Multiple Threads on a CMT System

Darryl Gove
Senior Staff Engineer
Sun Microsystems Inc.
Outline

• Multiple processes
• Simple problem
• fork - multi-process
• pthreads - multi-thread
• OpenMP - multi-thread
• Autopar - multi-thread
• Compiler options
• Supporting tools
## Options for parallelism

<table>
<thead>
<tr>
<th>Style</th>
<th>Task description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Less Collaboration</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Processes</td>
<td>Independent Heterogeneous</td>
<td>Consolidation</td>
</tr>
<tr>
<td>Multi-process</td>
<td>Independent Homogeneous</td>
<td>Web server</td>
</tr>
<tr>
<td>Pthreads</td>
<td>Shared tasks</td>
<td>Web browser</td>
</tr>
<tr>
<td><strong>More Communication</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>Shared tasks Splitting</td>
<td>Array Manipulation</td>
</tr>
<tr>
<td>Autopar</td>
<td>Shared tasks Splitting</td>
<td>Array Manipulation</td>
</tr>
</tbody>
</table>
Multiple processes

• Comparing
  > 6 core, 1 Ghz UltraSPARC-T1
  > 2x 750 Mhz UltraSPARC-III

• N copies of:

$ time gzip -c --best myfile | gunzip -c - >/dev/null
Multiple processes scaling

Throughput

- UltraSPARC-T1 (6 core, 1GHz)
- UltraSPARC-III (2x750MHz)

UltraSPARC-T1 over-subscribed
Scaling - zoom

Throughput

Heading towards two copies per core

UltraSPARC-T1 (6 core, 1GHz)
UltraSPARC-III (2x750MHz)
Observations

• Scaling beyond number of cores.
• How CMT is implemented is not important
• Even over-subscribed performance doesn't drop
A simple problem

```c
void main(int argc, const char** argv)
{
    int i;
    hrtime_t start_time;

    array=(int*)malloc(sizeof(int)*SIZE);
    int sum=0;

    for (i=0; i<SIZE; i++) { array[i]=1; }

    start_time=gethrtime();
    for (i=0; i<SIZE; i++)
    {
        sum+=array[i];
    }
    printf("Elapsed time (seconds)=\%5.3f ",
            (gethrtime()-start_time)/1000000000.0);
    printf("Total is \%i\n",sum);
}
```
Using fork()

```c
rtn=1;
start_time=gethrtime();
for (id=0; id<threads; id++)
{
    if (rtn != 0) {rtn=fork();} /*Fork main thread*/
    else {break;} /*Don't fork child*/
}

if (rtn==0)
{
    sum=0; /* Child thread*/
    for (i=(id-1)*array_length; i<id*array_length; i++)
    {
        sum+=array[i];
    }
    exit(0);
}
for (id=0; id<threads; id++) { wait(0); }
printf("Elapsed time (seconds)=%5.3f \n",
        (gethrtime()-start_time)/1000000000.0);
```
Scaling using fork()

Throughput - fork

Throughput

Threads

UltraSPARC-T1 (6 core, 1GHz)

UltraSPARC-III (2x750MHz)
Problems using fork() in this case

• (By default) No data shared between children
• Large footprint
• Lots of overhead
• Hard to pass results back. Solutions:
  > Message passing
  > Doors
  > Pipes
  > Shared memory
Using pthreads - create and join

```c
start_time=gethrtime();
for (id=0; id<threads; id++)
{
    pthread_create(&thread_array[id],NULL,&thread_code,(void*)id);
}

/*Join the threads*/
for (id=0; id<threads; id++)
    pthread_join(thread_array[id],NULL);

for (id=0; id<threads; id++)
    global_sum+=results[id];

printf("Elapsed time (seconds)=%5.3f ",
    (gethrtime()-start_time)/1000000000.0);
```
Using pthreads - doing work

```c
void* thread_code(void* v)
{
    int i;
    int sum = 0;
    int id = (int)v;
    for (i=(id)*array_length; i<(id+1)*array_length; i++)
    {
        sum+=array[i];
    }
    results[id]=sum;
    return 0;
}
```
Compiling

```
c -O -mt example.c -lpthread
```
Throughput - pthreads

Throughput

Threads

UltraSPARC-T1 (6 core, 1GHz)  UltraSPARC-III (2x750MHz)
Comments on pthreads

- Good scaling
- Control over activities of threads
- Significant coding overhead
False sharing

```c
void* thread_code(void* v)
{
    int i;
    int sum = 0;
    int id = (int)v;
    for (i=(id)*array_length; i<(id+1)*array_length; i++)
    {
        sum+=array[i];
    }
    results[id]=sum;
    return 0;
}
```

False sharing is when two or more threads try to update different parts of the same cacheline. Not called often - so not an issue.

- False sharing is when two or more threads try to update different parts of the same cacheline.
- Not called often - so not an issue.
False sharing - example

```c
volatile int * results;
...

void* thread_code(void* v)
{
    int i;
    int results[id] = 0;
    int id = (int)v;
    for (i=(id)*array_length; i<(id+1)*array_length; i++)
    {
        results[id]+=array[i];
    }
    /*results[id]=sum;  */
    return 0;
}
```

Threads update different variables on the same cacheline
Called frequently - so is an issue!

Two threads trying to update the same cacheline
False sharing performance

Throughput - pthreads with false sharing

UltraSPARC-T1 (6 core, 1GHz)  UltraSPARC-III (2x750MHz)
False sharing performance

Throughput - pthreads with false sharing

Throughput

Threads

UltraSPARC-T1 (6 core, 1GHz)

UltraSPARC-III (2x750MHz)
Comments on false sharing

- Multiple processors write to different parts of the same cacheline
- Causes significant performance impact on MP systems
- Avoid by padding structures to cacheline (64-byte) boundaries
- Not a problem for single chip CMT
Thread Local Storage - TLS

```c
int results=0;
pthread_mutex_t results_mutex;
__thread int sum = 0;
__thread int id;
...

void* thread_code(void* v)
{
    int i;
    int id = (int)v;
    for (i=(id)*array_length; i<(id+1)*array_length; i++)
    {
        sum+=array[i];
    }
    pthread_mutex_lock(&results_mutex);
    results += sum;
    pthread_mutex_unlock(&results_mutex);
    return 0;
}

void main()
{
    pthread_mutex_init(&results_mutex,NULL);
    ...}
```

Reduction
TLS Throughput On UltraSPARC-T1

Throughput of TLS code

Some problem with TLS code

Throughput

Threads

Local variable
Thread Local Storage
TLS - aliasing issue

```c
__thread int sum;
    int* array;

....

for (i=(id)*array_length; i<(id+1)*array_length; i++)
{
    sum+=array[i];
}

...
```

Sum “could be” an element in the array

```
10f88:  ac 05 a0 01  inc %l6
10f8c:  ba 07 40 05  add %i5, %g5, %i5
10f90:  fa 27 20 00  st %i5, [%i4] //Store sum 4, %l5
10f94:  aa 05 60 04  inc  %16, %17
10f98:  80 a5 80 17  cmp %icc, 0x10f88
10f9c:  24 4f ff fb  ble,a,pt [%15], %g5 //Load array
10fa0:  ca 05 60 00  ld  
...```
Restrict pointers

- Need to tell compiler that array does not point to `sum`
- The keyword `restrict` can be applied to pointer to mean that they point to their own memory
  ```c
  int * restrict array;
  ```
- Performance restored!
Thread Local Storage

- TLS mechanism to have “thread-global” variables
- Easier than setting up structures
- Non-local variables can potentially alias:
  - `restrict` keyword
  - `-xrestrict` compiler flag
  - `-xalias_level=<>` compiler flag
Mutex - making a scaling problem

```c
void* thread_code(void* v)
{
    int i;
    int id = (int)v;
    for (i=(id)*array_length; i<(id+1)*array_length; i++)
    {
        pthread_mutex_lock(&results_mutex);
        results+=array[i];
        pthread_mutex_unlock(&results_mutex);
    }

    return 0;
}
```

All loop iterations are mutexed
Mutex constrained scaling

Throughput with mutex limited scaling

Throughput

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Threads

0 1 2

UltraSPARC-T1 6 V880 8 cpus
cores 1GHz 1.05GHz
Mutex observations

• Using mutexes limits scaling
• Less impact on CMT processor
OpenMP

```c
void main(int argc, const char** argv)
{
    int i;
    hrtime_t start_time;

    array=(int*)malloc(sizeof(int)*SIZE);
    int sum=0;

    for (i=0; i<SIZE; i++) { array[i]=1; } 
    start_time=gethrtime();

#pragma omp parallel for reduction(+:sum)
    for (i=0; i<SIZE; i++)
    {
        sum+=array[i];
    }
    printf("Elapsed time (seconds)=%5.3f ",
            (gethrtime()-start_time)/1000000000.0);
    printf("Total is %i\n",sum);
}
```
OpenMP compiling and running

- Compile with:
  ```
  > cc -O -xopenmp example.c
  ```
- Run with:
  ```
  > export OMP_NUM_THREADS=2
  > a.out
  ```

Number of threads set in environment
OpenMP comments

- Same scaling as pthread version.
- OpenMP implemented on pthreads.
- Much easier to add: single line added to source file.
- Incremental parallelisation possible.
OpenMP: sleep or spin?

- Idle thread behaviour controlled by:
  
  SUNW_MP_THR_IDLE=SLEEP
  SUNW_MP_THR_IDLE=SPIN

- **SLEEP** idle threads consume no resources, so better for other processes on machine. (Current default is to sleep after short delay.)

- **SPIN** idle threads instantly ready to do work, but consume processor resources. Causes lower performance on over-subscribed systems.
OpenMP: binding to core

• Avoids process migration
• Can cause oversubscription
• Environment setting:
  > SUNW_MT_PROCBIND="true"
  > SUNW_MT_PROCBIND="0 2 4 6"
# Autopar: Original code

```c
void main(int argc,const char** argv)
{
    int i;
    hrtime_t start_time;

    array=(int*)malloc(sizeof(int)*SIZE);
    int sum=0;

    for (i=0; i<SIZE; i++) { array[i]=1; }

    start_time=gethrtime();

    for (i=0; i<SIZE; i++)
    {
        sum+=array[i];
    }

    printf("Elapsed time (seconds)=%5.3f ",
           (gethrtime()-start_time)/1000000000.0);
    printf("Total is %i\n",sum);
}
```
Compiling with autopar

• Compiler flags:
  cc -O -xautopar -xreduction -xloopinfo test.c

"test.c", line 24: not parallelized, unsafe dependence (array)
"test.c", line 30: PARALLELIZED, reduction

• -xautopar Enable autoparallelisation
• -xreduction Perform reduction optimisations
• -xloopinfo Report on parallelisation
Autopar: aliasing issue

• Problem:
  ```c
  int * array;
  ...
  for (i=0; i<SIZE;i++) { array[i]=1; }
  ```
  "example.c", line 24: not parallelized, unsafe dependence (array)

• Solutions:
  ```c
  int * restrict array;
  or
  compiler flag: -xalias_level=basic
  ```
Compiling and optimising

- Optimisation still important
  Total throughput = Single thread throughput * Threads
- Reduce instruction count
- **NEW**: Useless speculation steals from other threads
- So....
  > No big change
Compiling recap

• Always use optimisation (at least \(-O\))
• \(-xtarget=generic[64]\) for 1 binary to many processors
• Debug doesn't hurt (\(-g\), but \(-g0\) for C++)
• Crossfile optimisation (\(-xipo=2\)) particularly C++
• Use large pages (eg \(-xpagesize=64K\))
• Always profile the application
Profiling the mutex code

- Using Sun Studio Performance Analyzer
  
  collect <application> <arguments>

  analyzer test.1.er

<table>
<thead>
<tr>
<th>User CPU (sec.)</th>
<th>User CPU (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.401</td>
<td>1.401</td>
</tr>
<tr>
<td>0.140</td>
<td>23.947</td>
</tr>
<tr>
<td>0.020</td>
<td>0.020</td>
</tr>
<tr>
<td>0.120</td>
<td>8.676</td>
</tr>
</tbody>
</table>

Source File: ./mutex.c
Object File: ./mutex
Load Object: <mutex>

Loop could not be pipelined because it contains calls

30. for (i=(id)*array_length; i<(id+1)*array_length; i++)
31. {
32.   pthread_mutex_lock(&results_mutex);
33.   results+=array[i];
34.   pthread_mutex_unlock(&results_mutex);
35. }
36. return 0;

Cool tools for SPARC systems

http://cooltools.sunsource.net/

- GCC for SPARC Systems
- Simple Performance Optimisation Tool
- Automatic Tuning and Troubleshooting Tool
- ...

Darryl Gove - Coding for Multiple Threads on a CMT System - Sun Microsystems Inc - Multicore Expo 23rd March 2006
Concluding remarks

- Many ways to utilise a CMT processor
- Impact of traditional scaling limiters reduced on CMT - therefore easier to extract scaling opportunities
- Traditional compiler optimisation still works
- Issues are aliasing and locking
Coding for Multiple Threads on a CMT System.

Darryl Gove
darryl.gove@sun.com
http://blogs.sun.com/d/