Performance and Debugging Issues in OpenMP

Topics

- Scalable Speedup and Data Locality
- Parallelizing Sequential Programs
- Breaking data dependencies
- Avoiding synchronization overheads
- Achieving Cache and Page Locality
- Debugging

Factors impacting performance

- performance of single threaded code
- percentage of code that is run in parallel and scalability
- CPU utilization, effective data sharing, data locality and load balancing
- amount of synchronization and communication
- overhead to create, resume, manage, suspend, destroy and synchronize threads
- memory conflicts due to shared memory or falsely shared memory
- performance limitations of shared resources e.g memory, bus bandwidth, CPU execution units

Scalable Speedup

- Most often the memory is the limit to the performance of a shared memory program
- On scalable architectures, the latency and bandwidth of memory accesses depend on the locality of accesses
- In achieving good speedup of a shared memory program, data locality is an essential element

What Determines Data Locality

- In multi-node system, initial data distribution determines on which node the memory is placed
  - first touch or round-robin system policies
  - data distribution directives
  - explicit page placement
- Work sharing, e.g., loop scheduling, determines which thread accesses which data
- Cache friendliness determines how often main memory is accessed

Cache Friendliness

- For both serial loops and parallel loops
- locality of references
  - spatial locality: use adjacent cache lines and all items in a cache line
  - temporal locality: reuse same cache line; may employ techniques such as cache blocking
- low cache contention
  - avoid the sharing of cache lines among different objects; may resort to array padding or increasing the rank of an array

Cache Friendliness

- Contention is an issue specific to parallel loops, e.g., false sharing of cache lines

- cache friendliness =

  high locality of references

  +

  low contention

NUMA machines

- Memory hierarchies exist in single-CPU computers and Symmetric Multiprocessors (SMPs)
- Distributed shared memory (DSM) machines based on Non-Uniform Memory Architecture (NUMA) add levels to the hierarchy:
  - local memory has low latency
  - remote memory has high latency

Origin2000 memory hierarchy

Level	Latency (cycles)
register	0
primary cache	2..3
secondary cache	8..10
local main memory & TLB hit	75
remote main memory & TLB hit	250
main memory & TLB miss	2000
page fault	10^6

Page Level Locality

An ideal application has full page locality: pages accessed by a processor are on the same node as the processor, and no page is accessed by more than one processor (no page sharing)

Twofold benefit:

- » low memory latency
- » scalability of memory bandwidth

Performance Issues

Idle threads do no useful work

Divide work among threads as evenly as possible
- Threads should finish parallel tasks at same time

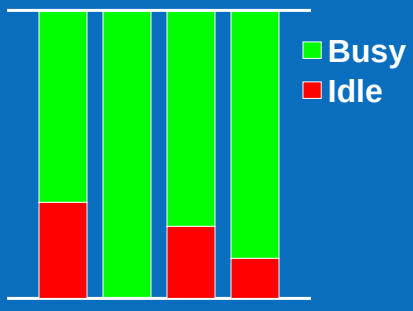
Synchronization may be necessary
- Minimize time waiting for protected resources

Load Imbalance

Unequal work loads lead to idle threads and wasted time.

```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

time



Performance Tuning

Profilers use sampling to provide performance data.

Traditional profilers are of limited use for tuning OpenMP*:

- Measure CPU time, not wall clock time
- Do not report contention for synchronization objects
- Cannot report load imbalance
- Are unaware of OpenMP constructs

Programmers need profilers specifically designed for OpenMP.

Parallelizing Code 1

- Optimize single-CPU performance
  - maximize cache reuse
  - eliminate cache misses
- Parallelize as high a fraction of the work as possible
  - preserve cache friendliness
-

Parallelizing Code 2

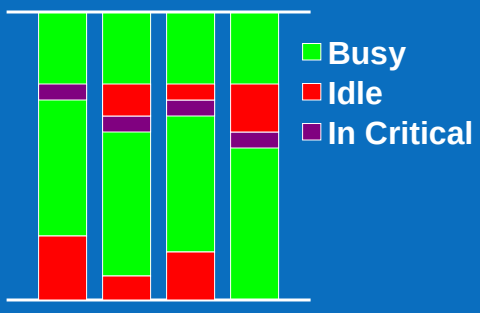
- Avoid synchronization and scheduling overhead:
  - partition in few parallel regions,
  - avoid reduction, single and critical sections,
  - make the code loop fusion friendly
  - use static scheduling
- Partition work to achieve load balancing
- Check correctness of parallel code
- Run OpenMP compiled code first on one thread, then on several threads

Synchronization

Lost time waiting for locks

```
#pragma omp parallel
{

#pragma omp critical
  {
    ...
  }
  ...
}
```

time



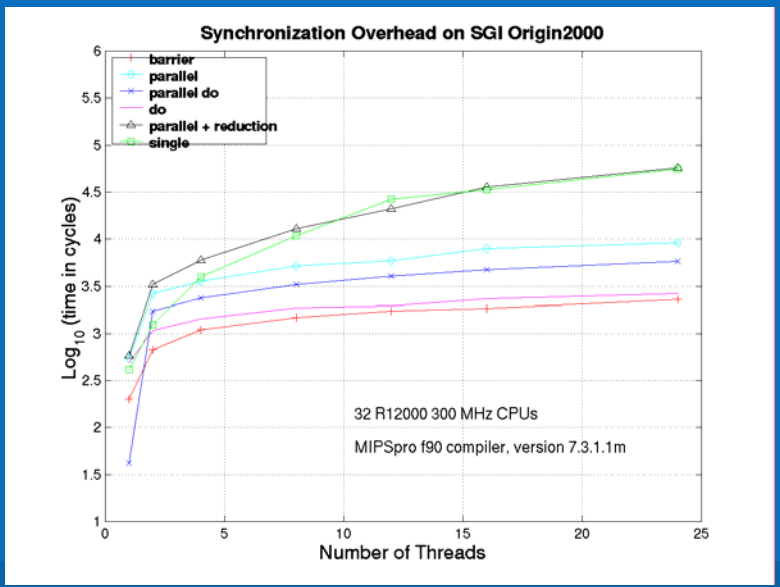
Synchronization Overhead

Parallel regions, work-sharing, and synchronization incur overhead

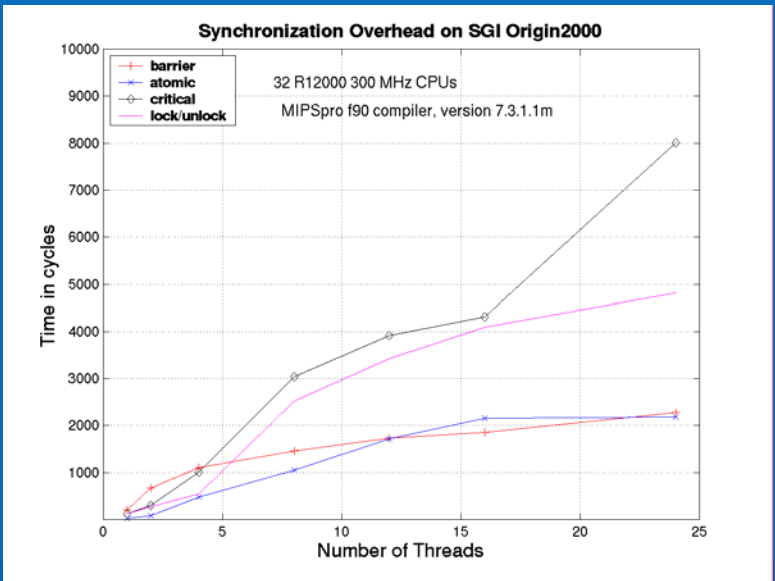
Edinburgh OpenMP **Microbenchmarks**, version 1.0, by J. Mark Bull,

In next slides used to measure the cost of synchronization on a 32 processor Origin 2000, with 300 MHz R12000 processors, and compiling the benchmarks with MIPSpro Fortran 90 compiler, version 7. 3.1.1m

Synchronization Overhead



Synchronization Overhead



Insights

cost (DO) ~ cost(barrier)

cost (parallel DO) ~ 2 * cost(barrier)

cost (parallel) > cost (parallel DO)

atomic is less expensive than critical

bad scalability for

- reduction
- mutual exclusion: critical, (un)lock
- single

Overhead on 4-way Intel Xeon at 3.0GHz Intel compiler and runtime library

Constructs	Cost (microsecs)	Scalability
parallel	1.5	linear
barrier	1.0	Linear or O(log(n))
Schedule (static)	1.0	linear
Schedule (guided)	6.0	Depend on contention
Schedule (dynamic)	50	Depend on contention
ordered	0.5	Depend on contention
single	1.0	Depend on contention
reduction	2.5	Linear or O(log(n))
atomic	0.5	Depend on data-type/hardware
Critical lock/unlock	0.5	Depend on contention

Overhead on 4-core Intel 2.4GHz gcc compiler and gomp runtime library

Constructs	Cost (microsecs)	Scalability
parallel	29.2	linear
barrier	18.9	Linear or O(log(n))
Schedule (static)	13.8	linear
Schedule (guided)	24.2	Depend on contention
Schedule (dynamic)	345.2	Depend on contention
ordered	4.2	Depend on contention
single	21.5	Depend on contention
reduction	29.4	Linear or O(log(n))
atomic	0.65	Depend on data-type/hardware
Critical lock/unlock	1.4	Depend on contention

Overhead on 2 processor Opteron gcc compiler and gomp runtime library

Constructs	Cost (microsecs)	Scalability
parallel	8.8	linear
barrier	4.0	Linear or O(log(n))
Schedule (static)	3.8	linear
Schedule (guided)	6.1	Depend on contention
Schedule (dynamic)	29.0	Depend on contention
ordered	4.8	Depend on contention
single	3.3	Depend on contention
reduction	9.4	Linear or O(log(n))
atomic	0.1	Depend on data-type/hardware
Critical lock/unlock	1.9	Depend on contention

Loop Parallelization

Identify the loops that are bottleneck to performance

Parallelize the loops, and ensure that

- no data races are created
- cache friendliness is preserved
- page locality is achieved
- synchronization and scheduling overheads are minimized

Hurdles to Loop Parallelization

- Data dependencies among iterations caused by shared variables
- Input/Output operations inside the loop
- Calls to thread-unsafe code, e.g., the intrinsic function rtc
- Branches out of the loop
- Insufficient work in the loop body

Data Races

Parallelizing a loop with data dependencies causes data races: unordered or interfering accesses by multiple threads to shared variables, which make the values of these variables different from the values assumed in a serial execution

A program with data races produces unpredictable results, which depend on thread scheduling and speed.

Types of Data Dependencies

Reduction operations:

```
const int n = 4096;
int a[n], i, sum = 0;
for (i = 0; i < n; i++)  {
  sum += a[i];
}
```

Easy to parallelize using reduction variables

Types of Data Dependencies

```
const int n = 4096;
int a[n], i, sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++)  {
   sum += a[i];
}
```

Types of Data Dependencies

Carried dependence on a shared array, e.g., recurrence:

```
const int n = 4096;
int a[n], i;
for (i = 0; i < n-1; i++)  {
  a[i] = a[i+1];
}
```

Non-trivial to eliminate

Parallelizing the Recurrence

Idea: Segregate even and odd indices

```
#define N 16384
int a[N], work[N+1];

// Save border element
work[N]= a[0];

// Save & shift even indices
#pragma omp parallel for
for ( i = 2;  i < N; i+=2)
{
  work[i-1] = a[i];
}
```

```
// Update even indices from odd
#pragma omp parallel for
for ( i = 0;  i < N-1; i+=2)
{
  a[i] = a[i+1];
}
// Update odd indices with even
#pragma omp parallel for
for ( i = 1;  i < N-1; i+=2)
{
  a[i] = work[i];
}
// Set border element
  a[N-1] = work[N];
```

Performing Reduction

The bad scalability of the reduction clause affects its usefulness, e.g., bad speedup when summing the elements of a matrix:

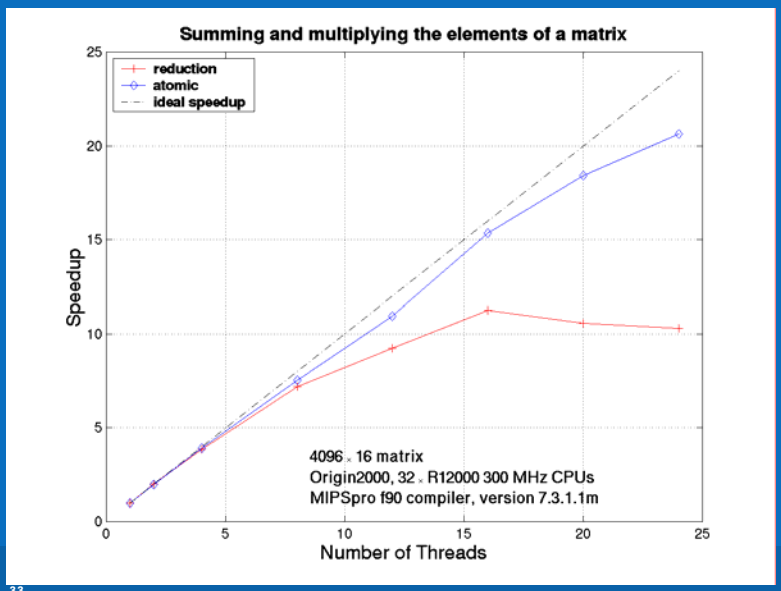
```
#define N 1<<12
#define M 16
int i, j;
double a[N][M], sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
        sum += a[i][j];
```


Parallelizing the Sum

Idea: Use explicit partial sums and combine them atomically

```
#define N 1<<12
#define M 16

int main() {
 double a[N][M], sum = 0.0;
 int i, j = 0;

#pragma omp parallel private(i,j)
 {
  double mysum = 0.0;
 // initialization of a
 // not shown

 // compute partial sum
#pragma omp for nowait
for (i = 0; i < N; i++)
 for (j = 0;  j < M; i++)
  mysum += a[i][j];
}

// each thread adds its
// partial sum
#pragma omp atomic
 sum += mysum;
  }
}
```

Sum and Product Speedup on SGI



Loop Fusion

- Recall that at the end of the parallel region, the threads are suspended and wait for the next parallel region, loop or section
  - Suspend/resume operations lighter weight than create/terminate but still create overhead
- Loop Fusion fuses loops to increase the work in the loop body
- Better serial programs: fusion promotes software pipelining and reduces the frequency of branches
- Better OpenMP programs: fusion reduces synchronization and scheduling overhead
  - fewer parallel regions and work-sharing constructs

Promoting Loop Fusion

- Loop fusion inhibited by statements between loops which may have dependencies with data accessed by the loops
- Promote fusion: reorder the code to get loops which are not separated by statements creating data dependencies
- Use one parallel do construct for several adjacent loops; may leave it to the compiler to actually perform fusion

Fusion-friendly code

Unfriendly

```
integer,parameter::n=4096
real :: sum, a(n)
do i=1,n
  a(i) = sqrt(dble(i*i+1))
enddo
sum = 0.d0
do i=1,n
  sum = sum + a(i)
enddo
```

Friendly

```
integer,parameter::n=4096
real :: sum, a(n)
sum = 0.d0
do i=1,n
  a(i) = sqrt(dble(i*i+1))
enddo
do i=1,n
  sum = sum + a(i)
enddo
```

Fusion-friendly code

Unfriendly

```
int n=4096;
double sum, a[4096];
for (i=0;i<n; i++) {
  a[i] = sqrt(double(i*i+1));
}
sum = 0.d0;
for (i=0;i<n; i++) {
   sum = sum + a[i];
}
```

Friendly

```
int n=4096;
double sum, a[4096];
sum = 0.d0;
for (i=0;i<n; i++) {
  a[i] = sqrt(double(i*i+1));
}

for (i=0;i<n; i++) {
   sum = sum + a[i];
}
```


Tradeoffs in Parallelization

- To increase parallel fraction of work when parallelizing loops, it is best to parallelize the outermost loop of a nested loop
- However, doing so may require loop transformations such as loop interchanges, which can destroy cache friendliness, e.g., defeat cache blocking
- Static loop scheduling in large chunks per thread promotes cache and page locality but may not achieve load balancing
- Dynamic and interleaved scheduling achieve good load balancing but cause poor locality of data references

Tuning the Parallel Code

- Examine resource usage, e.g., execution time, number of floating point operations, primary, secondary, and TLB cache misses and identify
  - the performance bottleneck
  - the routines generating the bottleneck
- Correct the performance problem and verify the desired speedup.

The Future of OpenMP

- Data placement directives will become part of OpenMP
  - affinity scheduling may be a useful feature
- It is desirable to add parallel input/output to OpenMP
- Java binding of OpenMP

Debugging OpenMP programs

- Standard debuggers do not normally handle OpenMP
- approach :

1. use binary search to try to narrow down where the problem is by disabling OpenMP pragmas
2. Compile with –fopenmp_stubs if available – this lets one run a serial version. If the bug persists it is in the serial code so debug as a serial program
3. Compile with –fopenmp and OMP_NUM_THREADS=1. If it still fails debug in single threaded mode.
4. Identify the errors with the lowest optimization possible

Debugging OpenMP programs

1. use binary search to try to narrow down where the problem is by disabling OpenMP pragmas
2. Compile with –fopenmp_stubs if available – this lets one run a serial version. If the bug persists it is in the serial code so debug as a serial program
3. Compile with –fopenmp and OMP_NUM_THREADS=1. If it still fails debug in single threaded mode.
4. Identify the errors with the lowest optimization possible
5. Look for problems such as data dependence, race conditions, deadlock, missing barriers, unitialized variables
6. Compile using a thread checker if available

References

Introduction to OpenMP

Lawrence Livermore National Laboratory

www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html

Ohio Supercomputing Center

oscinfo.osc.edu/training/openmp/big

Minnesota Supercomputing Institute

www.msi.umn.edu/tutorials/shared_tutorials/openMP

References

OpenMP Benchmarks

Edinburgh OpenMP Microbenchmarks

www.epcc.ed.ac.uk/research/openmpbench

SPMD Example

A single parallel region, no scheduling needed, each thread explicitly determines its work

```
program mat_init
implicit none
integer, parameter::N=1024
real A(N,N)
integer :: iam, np
iam = 0
np = 1
!$omp parallel private(iam,np)
 np = omp_get_num_threads()
 iam = omp_get_thread_num()
! Each thread calls work
 call work(N, A, iam, np)
!$omp end parallel
end
```

```
subroutine work(n, A, iam, np)
integer n, iam, n
real A(n,n)
integer :: chunk,low,high,i,j
chunk = (n + np - 1)/np
low = 1 + iam*chunk
high=min(n,(iam+1)*chunk)
do j = low, high
   do I=1,n
  A(I,j)=3.14 + & sqrt(real(i*i*i+j*j+i*j*j))
 enddo
enddo
return
```


Pros and Cons of SPMD

- Potentially higher parallel fraction than with loop parallelism
- The fewer parallel regions, the less overhead
- More explicit synchronization needed than for loop parallelization
- Does not promote incremental parallelization and requires manually assigning data subsets to threads

Message passing vs multithreading

- Process versus thread address space
  - threads have shared address space, but the thread stack holds thread-private data
  - processes have separate address spaces
- For message passing multiprocessing, e.g., MPI, all data is explicitly communicated, no data is shared
- For OpenMP, threads in a parallel region reference both private and shared data
- Synchronization: explicit or embedded in communication