Brief Introduction to OpenMP

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

October 11, 2007

Background

- OpenMP is a framework for shared memory parallel computing.
- OpenMP is a standard C/C++ and Fortran compilers.
- Compiler directives indicate where parallelism should be used.
  - C/C++ use `#pragma` directives
  - Fortran uses structured comments.
- A library provides support routines.
- Based on the fork/join model:
  - the program starts as a single thread
  - at designated *parallel regions* a pool of threads is formed
  - the threads execute in parallel across the region
  - at the end of the region the threads wait for all of the team to arrive
  - the *master thread* the continues until the next parallel region.

Brief Introduction to OpenMP

- Some advantages:
  - Usually can arrange so the same code can run sequentially.
  - Can add parallelism incrementally.
  - Compiler can optimize.
- The OpenMP standard specifies support for C/C++ and Fortran.
- Many compilers now support OpenMP, for example
  - newer versions of `gcc` and `gfortran`
  - Intel compilers `icc` and `ifort`
- The OpenMP runtime creates and manages separate threads.
- OpenMP is much easier to use than low level thread libraries.
- You still have to make sure what you are doing is thread-safe.

A Simple Example

A very simple parallel C program:

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

main() {
#pragma omp parallel
    printf("Greetings!\n");
}
```

The same program in Fortran:

```
      program ompgreetings0
!$omp parallel
      print *,"Greetings!"
!$omp end parallel
      stop
      end
```

Fortran (usually) requires `end parallel` directives.

A Simple Example

Compile with

```
gcc4 -o ompgreetings0 -fopenmp ompgreetings0.c -lgomp
```

or with

```
gfortran -o ompgreetings0 -fopenmp ompgreetings0.f90 -lgomp
```

A sample run:

```
[luke@node00 ~]$ ./ompgreetings0
 Greetings!
 Greetings!
 Greetings!
 Greetings!
 Greetings!
 Greetings!
 Greetings!
 Greetings!
[luke@node00 ~]$
```

A Simple Example

- The default number of OMP threads on `beowulf` is currently set to the number of available cores.
- You can change this with the environment variable `OMP_NUM_THREADS`:

```
[luke@node00 ~]$ env OMP_NUM_THREADS=3 ./ompgreetings0
 Greetings!
 Greetings!
 Greetings!
[luke@node00 ~]$
```

- You can also change in in the OMP directive in the program:

```
#pragma omp parallel num_threads(3)
```

- The number of threads can also be an integer variable.

Some OMP Library Functions

- The OMP library provides a number of useful routines.
- Some of the most commonly used:
  - `omp_get_thread_num`: current thread index (0, 1, . . .)
  - `omp_get_num_threads`: size of the active team
  - `omp_get_max_threads`: maximum number of threads
  - `omp_get_num_procs`: number of processors available
  - `omp_get_wtime`: elapsed wall clock time from "some time in the past."
  - `omp_get_wtick`: timer resolution
- A variant of the greeting example using `omp_get_thread_num`:

```
#pragma omp parallel
    {
        int n = omp_get_thread_num();
        printf("Greetings from thread %d!\n", n);
    }
    printf("Greetings from the main thread!\n");
```

  The complete example is in `ompgreetings.c`.

Parallel Loops in OpenMP

- OpenMP provides directives to support parallel loops.
- The full version:

```
#pragma omp parallel
#pragma omp for
    for (i = 0; i < n; i++)
        ...
```

- Abbreviated versions:

```
#pragma omp parallel for
    for (i = start; i < end; i++)
        ...
```

- There are some restrictions on the loop, including:
  - The loop has to be of this simple form with
    - `start` and `end` computable before the loop
    - a simple comparison test
    - a simple increment or decrement expression
  - exits with `break`, `goto`, or `return` are not allowed.

Shared and Private Variables

- Variables declared before a parallel block can be *shared* or *private*.
- Shared variables are shared among all threads.
- Private variables vary independently within threads
  - On entry, values of private variables are undefined.
  - On exit, values of private variables are undefined.
- By default,
  - all variables declared outside a parallel block are shared
  - except the loop index variable, which is private
- Variables declared in a parallel block are always private
- Variables can be explicitly declared shared or private.

Shared and Private Variables

- A simple example:

```
#pragma omp parallel for
    for (i = 0; i < n; i++)
        x[i] = x[i] + y[i];
```

- Here `x`, `y`, and `n` are shared and `i` is private in the parallel loop.
- We can make the attributes explicit with

```
#pragma omp parallel for shared(x, y, n) private(i)
    for (i = 0; i < n; i++)
        x[i] = x[i] + y[i];
```

or

```
#pragma omp parallel for default(shared) private(i)
    for (i = 0; i < n; i++)
        x[i] = x[i] + y[i];
```

- The value of `i` is undefined after the loop.

Simple Parallel Matrix Multiply

- C version is in `matmult.c`
- Fortran version is in `matmult.f90`
- Uses parallel loop directives:
  - In C,

```
#pragma omp parallel for default(shared) private(i, j, n)
    for (j = 0; j < NY; j++)
        for (n = 0; n < NM; n++)
            for (i = 0; i < NX; i++)
                M(i, j) = M(i, j) + A(i, n) * B(i, j);
```

  - In Fortran,

```
!$omp parallel do default(shared) private(i, j, n)
   do j=1,ny
      do n=1,nm
         do i=1,nx
            m(i,j) = m(i,j) + a(i,n)*b(n,j)
         end do
      end do
   end do
!$omp end parallel
```

Critical Sections and Reduction Variables

- Suppose we want to parallelize

```
int sum = 0;
for (i = 0; i < n; i++) {
    int val = f(i);
    sum = sum + val;
}
```

- A first attempt:

```
    int sum = 0;
#pragma omp parallel for
    for (i = 0; i < n; i++) {
        int val = f(i);
        sum = sum + val;
    }
```

- Problem: there is a *race condition* in the updating of `sum`.

Critical Sections and Reduction Variables

- One solution is to use a critical section:

```
    int sum = 0;
#pragma omp parallel for
    for (i = 0; i < n; i++) {
        int val = f(i);
#pragma omp critical
        sum = sum + val;
    }
```

- Only one thread at a time is allowed into a critical section.
- An alternative is to use a *reduction variable*:

```
    int sum = 0;
#pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        int val = f(i);
        sum = sum + val;
    }
```

- Reduction variables are in between private and shared variables.
- Other supported reduction operators include `*`, `&&`, and `||`.

Some Additional Clauses

- `firstprivate`, `lastprivate` declare variables private.
- `firstprivate` variables are initialized to their value before the parallel section.
- For `lastprivate` variables the value of the variable after the loop is the value after the logically last iteration.
- Variables can be listed both as `firstprivate` and `lastprivate`.
- The `if` clause can be used to enable parallelization conditionally.
- `num_threads(P)` says to use `P` threads.
- `schedule(static, n)` divides the loop into chunks of size `n` assigned cyclically to the threads.
- `schedule(dynamic, n)` divides the loop into chunks of size `n` assigned cyclically to the next available thread.

Parallelizing Vectorized Operations in R

Code to evaluate a single argument function `f`:

```
#pragma omp parallel for if (P > 1) num_threads(P) \
        default(shared) private(i) reduction(||:nflag)
    for (i = 0; i < n; i++) {
        int ia = i;
        double ai = a[ia];
        if (ISNA (ai)) y[i] = NA_REAL;
        else if (ISNAN(ai)) y[i] = R_NaN;
        else {
            y[i] = f(ai);
            if (ISNAN(y[i])) naflag = 1;
        }
    }
```

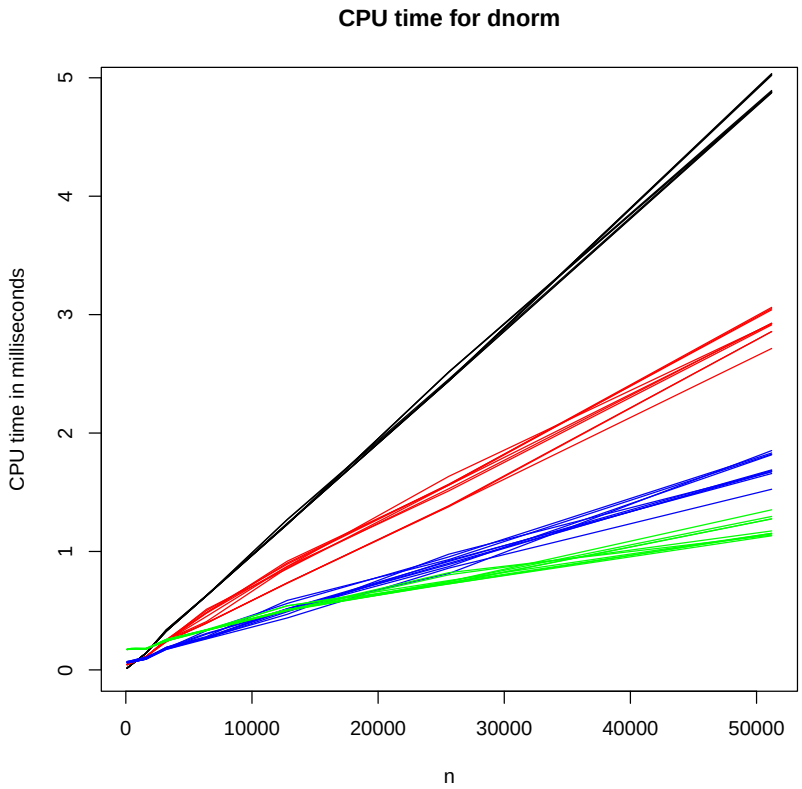
An alternative that may be useful for load balancing:

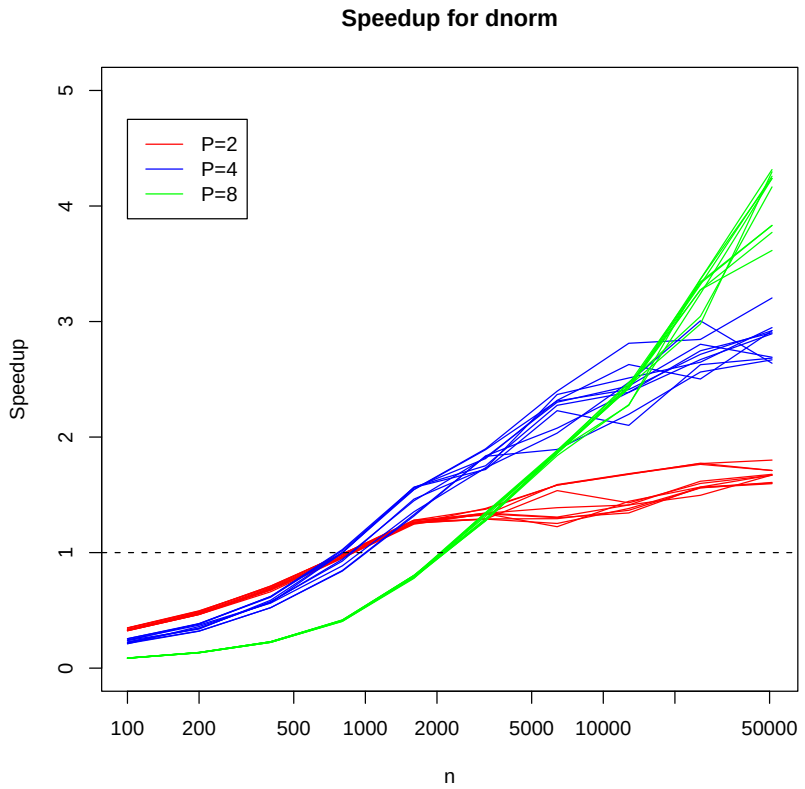
```
chunk = ...
#pragma omp parallel for if (chunk < n) schedule(dynamic, chunk) \
        default(shared) private(i) reduction(||:nflag)
```

More experimentation is needed.

Parallelizing Vectorized Operations in R

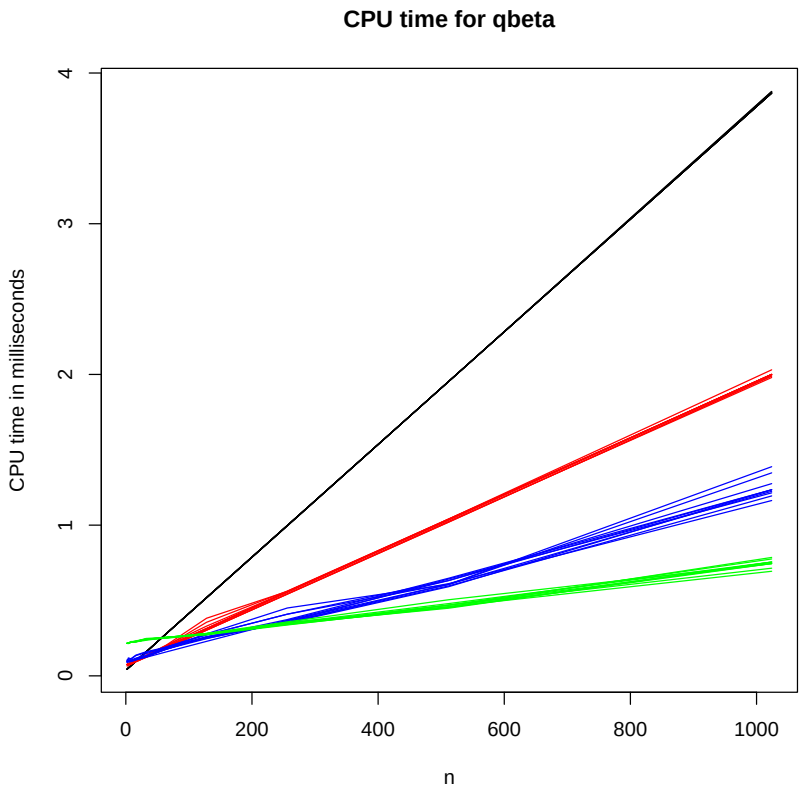
CPU times and speedup for `dnorm`:

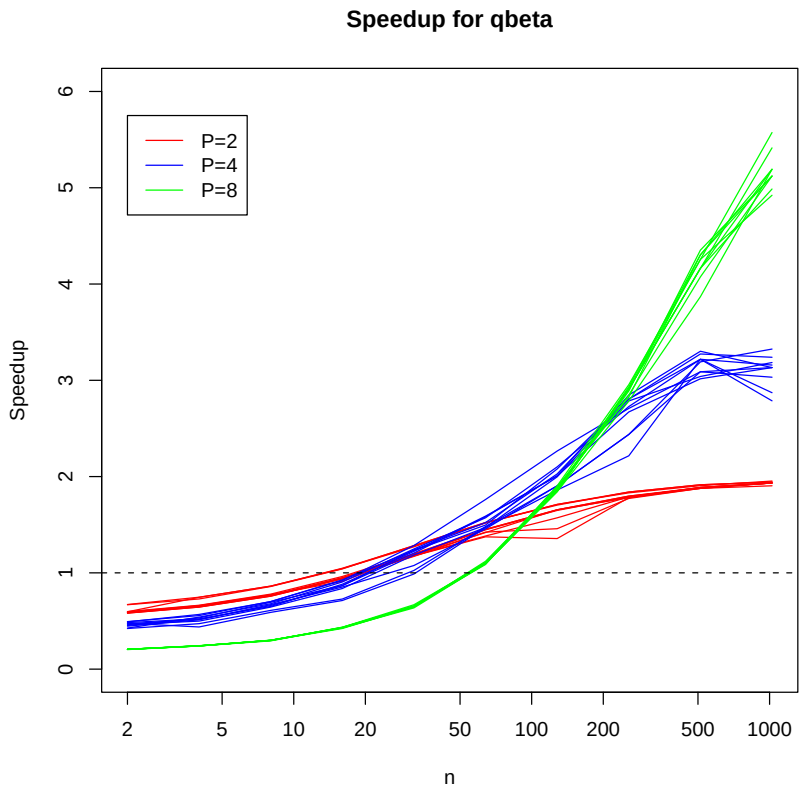




Parallelizing Vectorized Operations in R

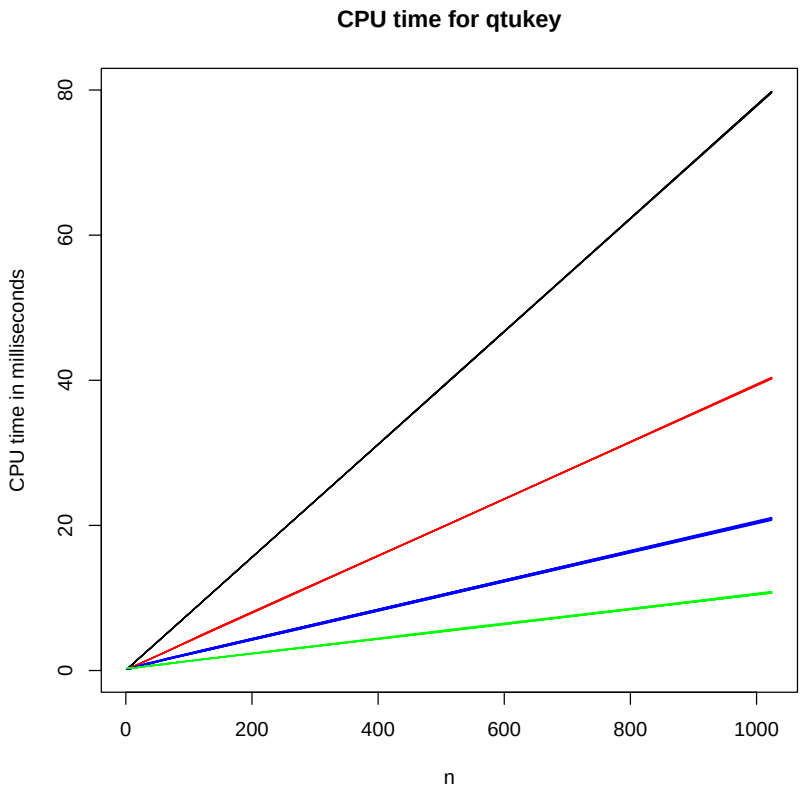
CPU times and speedup for `qbeta`:

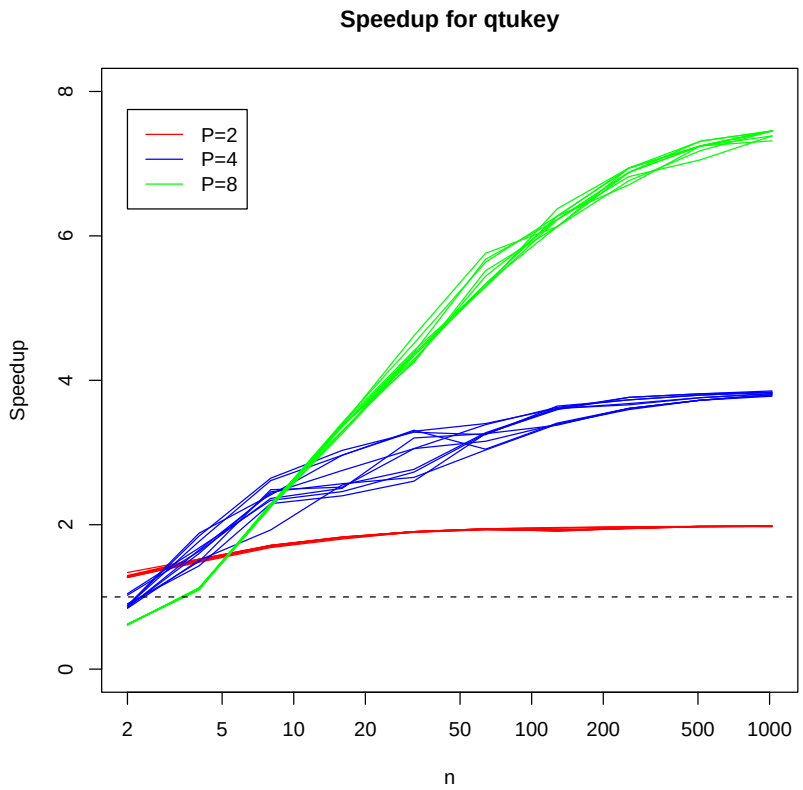




Parallelizing Vectorized Operations in R

CPU times and speedup for `qtukey`:





Parallelizing Vectorized Operations in R

- Machine used for these results:
  - Macintosh running OS X 10.4
  - Intel processors running i386 code.
  - Two quad-core processors.

  All of these may affect performance.
- For slow functions like `qtukey` parallelism pays immediately.
- For fast functions like `dnorm` overhead matters:
  - starting and stopping the threads
  - fetching and storing the data
- Choosing the right threshold for going parallel is important.
- Figuring out how to determine the thresholds effectively is hard.
- Load balancing may also be useful.
- An experimental package will hopefully be available in a month or so.

Intel Compilers and Automatic Parallelization

- Can the compiler insert OpenMP directives for you?
- Some compilers can; this is one form of *automatic parallelization*.
  - They have to be conservative.
  - Programs spread over several files create difficulties.
- The Intel compilers have some support for automatic parallelization.
- Compilers are `icc` and `ifort`.
- Option `-openmp` enables OpenMP processing.
- Option `-parallel` enables automatic parallelization.
- Option `-par-report3` is useful to see what it does.
- Automatic parallelization is easier/works better in Fortran.
- Example commands for automatic parallelization:

```
ifort -O3 -parallel -par-report3 matmult.f90  -o matmult_f
icc -O3 -parallel -par-report3 -openmp matmult.c -o matmult_c
```

Automatic Parallelization of C Code

```
procedure: main
serial loop: line 21: not a parallel candidate due to insufficent work
serial loop: line 27: not a parallel candidate due to insufficent work
serial loop: line 34: not a parallel candidate due to insufficent work
serial loop: line 33: not a parallel candidate due to insufficent work
serial loop: line 43: not a parallel candidate due to insufficent work
serial loop: line 20
   output data dependence assumed from line 22 to line 22, due to "a"
serial loop: line 26
   output data dependence assumed from line 28 to line 28, due to "b"
serial loop: line 41
   anti data dependence assumed from line 44 to line 44, due to "m"
   output data dependence assumed from line 44 to line 44, due to "m"
   flow data dependence assumed from line 44 to line 44, due to "m"
serial loop: line 42
   anti data dependence assumed from line 44 to line 44, due to "m"
   output data dependence assumed from line 44 to line 44, due to "m"
   flow data dependence assumed from line 44 to line 44, due to "m"
```

Automatic Parallelization of Fortran Code

```
   procedure: matmul_omp
   serial loop: line 18: not a parallel candidate due to insufficent work
   ...
   serial loop: line 42: not a parallel candidate due to insufficent work
matmult.f90(17) : (col. 4) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 17
      shared     : { "a" }
      private    : { "n" "i" }
      first priv.: { }
      reductions : { }
matmult.f90(24) : (col. 4) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 24
      ...
matmult.f90(40) : (col. 4) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 40
      shared     : { "a" "b" "m" }
      private    : { "n" "i" "j" }
      first priv.: { }
      reductions : { }
```