# Introduction to OpenMP with C++/C: Shared-memory multiprocessing

Greg Teichert

Consulting for Statistics, Computing and Analytics Research (CSCAR)

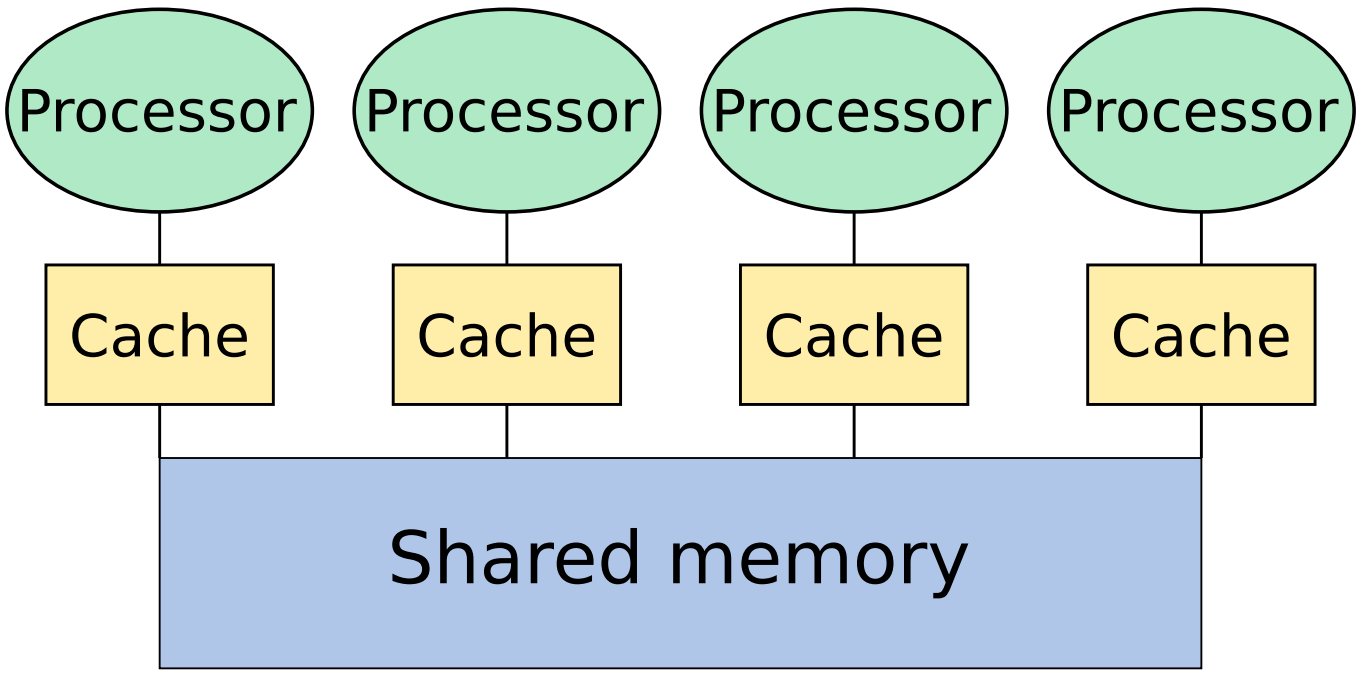
Thursday, 9/13, 2-4 pm

# Resources

- Slides available at github.com/greght/Workshop-OpenMP
- en.wikipedia.org/wiki/OpenMP
- www.openmp.org/resources/
- www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- Examples compiled, run, and saved using the online C++ compiler at coliru.stacked-crooked.com

# Shared memory parallel

- Each processor/core has access to shared memory.
- Each processor/core has a local cache where it performs operations.
- Values are copied between the local cache and main memory.



# OpenMP

- OpenMP: Open Multi-Processing
- An API for writing shared-memory parallel applications
- C, C++, Fortran
- openmp.org

# #pragma directive

- Directives give instructions to the compiler
- Examples:

```
#include <omp.h>
#define pi 3.14159
```

- #pragma (from "pragmatic") prefaces additional compiler instructions (accompanied by omp for OpenMP constructs):

```
#pragma omp [command]
```

- en.wikipedia.org/wiki/Directive_(programming)

# Creating threads with `parallel`

- "Hello, world!" example with OpenMP:

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

int main(){

  #pragma omp parallel
  {
    std::cout << "Hello, world!\n";
  }
  return 0;

}
```

- http://coliru.stacked-crooked.com/a/05ae3f19f728c6a7
- Possible output:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

# A few OpenMP functions...

- `omp_set_num_threads(N)`: set the number of threads to N
- `omp_get_num_threads()`: get the number of threads being used
- `omp_get_thread_num()`: get the thread numnber for the current thread

# A few OpenMP functions (example)

- Example:

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

int main(){

  omp_set_num_threads(3);
  #pragma omp parallel
  {
    int i, N;
    i = omp_get_thread_num();
    N = omp_get_num_threads();
    printf("This is thread %d of %d.\n",i,N);
  }
  return 0;

}
```

- http://coliru.stacked-crooked.com/a/06758d42dafe1256
- Possible output:

```
This is thread 2 of 3.
This is thread 1 of 3.
This is thread 0 of 3.
```

# Exercise 1: Scalar multiplication of a vector

- Perform scalar multiplication of a vector.
- Serial program:

```
#include <iostream>
#include <vector>

int main(){

  //Scalar multiplication of a vector

  //Initialize vector of length 10 with entries of 1.2
  std::vector<double> vec(10,1.2);
  for (unsigned int i=0; i<vec.size(); ++i){
    vec[i] *= 2.; //Multiply each entry by 2.
  }
  //Serially output values to double-check
  for (unsigned int i=0; i<vec.size(); ++i){
    std::cout << vec[i] << " ";
  }

  return 0;
}
```

- http://coliru.stacked-crooked.com/a/e9db2832a977953e
- Output:

```
 2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4
```

# Possible parallel solution

- Distribute the "for" loop over all threads.

```
#include <omp.h>
#include <iostream>
#include <vector>
```

...

```
  std::vector<double> vec(10,1.2);
  #pragma omp parallel
  {
    unsigned int j, n_thrds, N;
    j = omp_get_thread_num();
    n_thrds = omp_get_num_threads();
    N = vec.size();
    for (unsigned int i=(j*N)/n_thrds; i<((j+1)*N)/n_thrds; ++i)
     {
      vec[i] *= 2.; //Multiply each entry by 2.
    }
  }
```

...

- http://coliru.stacked-crooked.com/a/2859a33f5d6f5e4a
- Output:

```
2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4 2.4
```

# parallel for

OpenMP can distribute the "for" loop for you:

`#pragma omp for`

```
std::vector<double> vec(10,1.2);
#pragma omp parallel
{
  #pragma omp for
  for (unsigned int i=0; i<vec.size(); ++i){
    vec[i] *= 2.; //Multiply each entry by 2.
  }
}
```

Or, equivalently,

```
std::vector<double> vec(10,1.2);
#pragma omp parallel for
for (unsigned int i=0; i<vec.size(); ++i){
  vec[i] *= 2.; //Multiply each entry by 2.
}
```

http://coliru.stacked-crooked.com/a/467d669b9617f223

# Exercise 2: Calculation of vector norm

- Calculate the 2-norm of a vector: $|v| = \sqrt{\sum_i v_i^2}$.
- Serial program:

```
#include <iostream>
#include <vector>
#include <cmath>

int main(){

  //2-norm of a vector
  //Initialize vector of length 100,000,000 with entries of 3.2.
  std::vector<double> vec(100000000,3.2);
  double norm = 0.;
  for (unsigned int i=0; i<vec.size(); ++i){
    norm += vec[i]*vec[i];
  }
  norm = std::sqrt(norm);
  std::cout << norm << std::endl;

  return 0;
}
```

- http://coliru.stacked-crooked.com/a/5c683b0640d49f75
- Output:

```
32000
```

- Computation time: ∼1.3 sec

# Exercise 2a: INCORRECT implementation

- It is incorrect to simply apply the `parallel for` (but go ahead and do it to see what happens).
- Threads will try to update the global variable `norm` at the same time, becoming a "data race" (see next slide):

```
//Initialize vector of length 100,000,000 with entries of 3.2.
std::vector<double> vec(100000000,3.2);
double norm = 0.;
#pragma omp parallel for
for (unsigned int i=0; i<vec.size(); ++i){
  norm += vec[i]*vec[i];
}
norm = std::sqrt(norm);
```

- http://coliru.stacked-crooked.com/a/ab3a1044de29a5d0
- Possible outputs:

```
28988.9
```

```
28840.8
```

- Computation time: $\sim$2.7 sec on 2 processors

# Data race → inaccuracies

- Data race: two or more threads try to modify the same location at memory at the same time.
- Example (from en.wikipedia.org/wiki/Race_condition):

**Correct:**

| Thread 1 | Thread 2 | | Global Value |
|---|---|---|---|
| | | | 0 |
| Read (0) | | ← | 0 |
| Increase value (1) | | | 0 |
| Write (1) | | → | 1 |
| | Read (1) | ← | 1 |
| | Increase value (2) | | 1 |
| | Write (2) | → | 2 |

**Data race:**

| Thread 1 | Thread 2 | | Global Value |
|---|---|---|---|
| | | | 0 |
| Read (0) | | ← | 0 |
| | Read (0) | ← | 0 |
| Increase value (1) | | | 0 |
| | Increase value (1) | | 0 |
| Write (1) | | → | 1 |
| | Write (1) | → | 1 |

# Exercise 2b: Avoiding data races

- Calculate the 2-norm of a vector with OpenMP, while avoiding data races.
- Hint:

# Exercise 2b: Avoiding data races

- Calculate the 2-norm of a vector with OpenMP, while avoiding data races.
- Hint: use a vector of length `omp_get_num_threads()` to store the contribution from each thread.

# Exercise 2b: Accurate, but slow implementation

- Write to components of a global vector:

```
//Initialize vector of length 100,000,000 with entries of 3.2
std::vector<double> vec(100000000,3.2);
double norm = 0.;
unsigned int n_threads = 2;
omp_set_num_threads(n_threads);
std::vector<double> thread_sum(n_threads,0.);

#pragma omp parallel
{
  //Store the sum from each thread in an array
  unsigned int j = omp_get_thread_num();
  #pragma omp for
  for (unsigned int i=0; i<vec.size(); ++i){
    thread_sum[j] += vec[i]*vec[i];
  }
}
//Sum the contributions from each thread
for (unsigned int i=0; i<n_threads; ++i){
  norm += thread_sum[i];
}
norm = std::sqrt(norm);
```

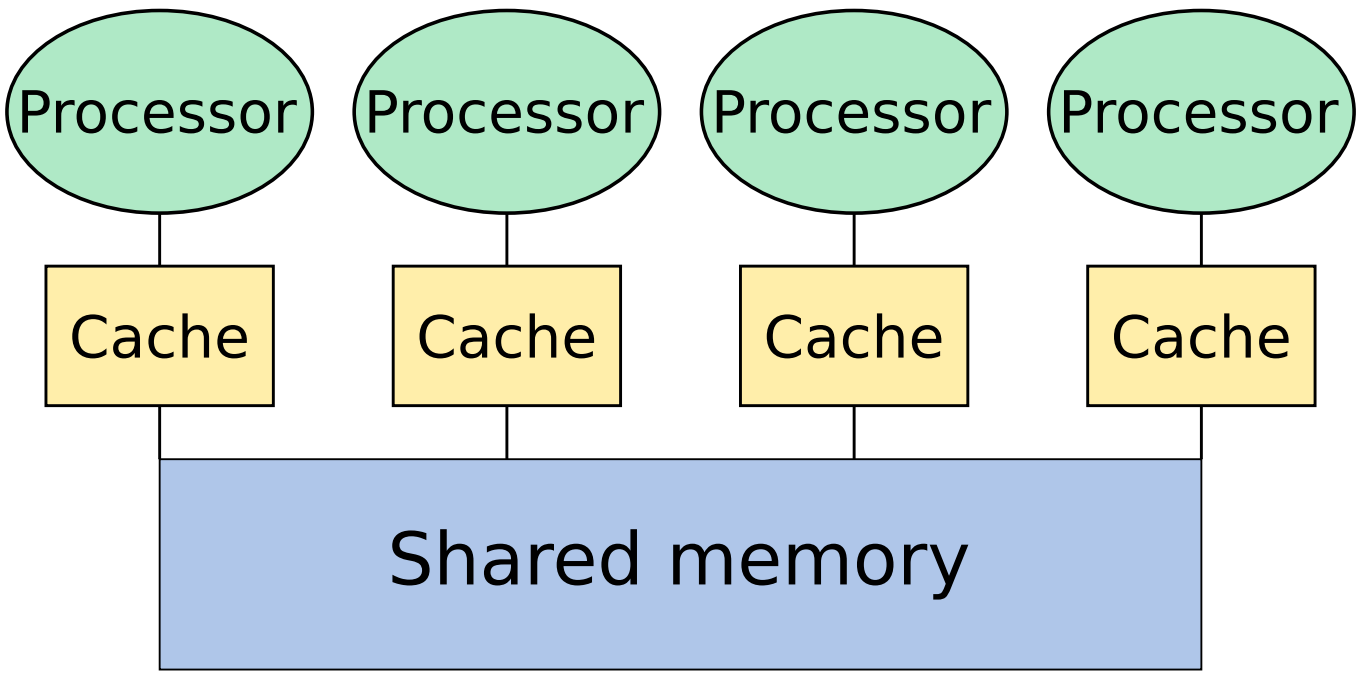
- http://coliru.stacked-crooked.com/a/41f1d832ac84ae1b
- Output:

```
32000
```

- Computation time: ∼3.1 sec on 2 processors

# False sharing → inefficiencies

- Each processor/core works with variables in the local cache.
- When a shared variable/array is updated by a different thread, the value is updated from shared memory for all threads.
- **This means that one thread updating one component of an array/vector will cause all threads to reload the entire array/vector.**
- The repeated reloading slows down performance.



# Exercise 2c: Avoiding data races and false sharing

- Calculate the 2-norm of a vector with OpenMP, while avoiding data races and false sharing.
- Hint:

# Exercise 2c: Avoiding data races and false sharing

- Calculate the 2-norm of a vector with OpenMP, while avoiding data races and false sharing.
- Hint: use a temporary double within each thread to sum within the "for" loop, then add to the vector as before.

# Exercise 2c: Fast, accurate implementation (with vector)

- Write to temporary variable in loop, use global vector to go from threads to global value:
  (...)

```
#pragma omp parallel
{
  //Store the sum from each thread in a temporary variable
  unsigned int j = omp_get_thread_num();
  double tmp = 0.;
  #pragma omp for
  for (unsigned int i=0; i<vec.size(); ++i){
    tmp += vec[i]*vec[i];
  }
  //Pass the temporary variable into the array
  thread_sum[j] = tmp;
}
//Sum the contributions from each thread
for (unsigned int i=0; i<n_threads; ++i){
  norm += thread_sum[i];
}
norm = std::sqrt(norm);
```

- http://coliru.stacked-crooked.com/a/cd2ff8c4cac1df88
- Output:

```
32000
```

- Computation time: ∼0.54 sec on 2 processors

# critical and atomic

- The directive `#pragma omp critical` allows only one thread to perform *any* "critical" operation at a time, e.g.:

```
#pragma omp parallel for
for (unsigned int i=0; i<100; ++i){
  #pragma omp critical
  if (sum < 25)
    sum += 1;
  //Only one thread can perform an operation listed in this or
    any critical section at a time
}
```

- The directive `#pragma omp atomic` is similar, but only applies to updating a memory location, e.g.:

```
#pragma omp parallel for
for (unsigned int i=0; i<100; ++i){
  #pragma omp atomic
  sum += 1;
  //Only one thread will read then update the global variable
   at a time
}
```

# Exercise 2d: Use `critical` or `atomic`

- Calculate the 2-norm of a vector with OpenMP, using `critical` or `atomic` (decide which is better) instead of the temporary vector/array.

# Exercise 2d: Fast, accurate, a little cleaner (with atomic)

- We can use either here, but *atomic* is most appropriate for this situation, since it is just updating the global variable.
- Note that it is still faster to use a temporary variable within the "for" loop, then add to the global variable.

```
//Initialize vector of length 100,000,000 with entries of 3.2
std::vector<double> vec(100000000,3.2);
double norm = 0.;
#pragma omp parallel
{
  double tmp = 0.;
  #pragma omp for
  for (unsigned int i=0; i<vec.size(); ++i){
    tmp += vec[i]*vec[i];
  }
  #pragma omp atomic
  norm += tmp;
}
norm = std::sqrt(norm);
```

- http://coliru.stacked-crooked.com/a/48197f623c823728
- Output:

```
10119.3
```

- Computation time: ∼0.57 sec on 2 processors

# reduction

- The process of combining values from multiple variables into a single variable (e.g. by summing or multiplying) is called **reduction**.
- This is done by specifying the operator and the accumulating variable with the clause: `reduction(op:var)`
- For example, summing the components of a vector:

```
std::vector<double> vec(10000000,3.2);
double sum = 0.;
#pragma omp parallel for reduction(+:sum)
for (unsigned int i=0; i<vec.size(); ++i){
  sum += vec[i];
}
```

# Exercise 2e: Use `reduction`

- Calculate the 2-norm of a vector with OpenMP, using `reduction`.

# Exercise 2e: "Best" implementation (with `reduction`)

- Code:

```
//Initialize vector of length 100,000,000 with entries of 3.2
std::vector<double> vec(100000000,3.2);
double norm = 0.;
#pragma omp parallel for reduction(+:norm)
for (unsigned int i=0; i<vec.size(); ++i){
  norm += vec[i]*vec[i];
}
norm = std::sqrt(norm);
```

- http://coliru.stacked-crooked.com/a/5760ab8c5c0ed51d
- Output:

```
32000
```

- Computation time: $\sim$0.58 sec on 2 processors

# sections

- If there are different tasks that can be done at the same time, the tasks can be done simultaneously using `sections`:

```
#pragma omp parallel
{
  //Call function1, function2, and function3 simultaneously
  #pragma omp sections
  {
    #pragma omp section
    //Call function1
    function1()

    #pragma omp section
    //Call function2
    function2()

    #pragma omp section
    //Call function3
    function3()
  }
}
```

# Exercise 3: Solve two initial value ODEs

$$\frac{dy_1}{dt} - y_1 = t + 1 \qquad y_1(0) = 0$$
$$\frac{dy_2}{dt} + y_2 = t^2 \qquad y_2(0) = 0.5$$

- Initial value problems generally use serial time-stepping schemes, i.e. the loop over time cannot be parallelized.
- Use `sections` to speed up the code.

# Exercise 3: Solve two initial value ODEs

- Serial code:

```
double dt = 0.000001;
unsigned int N=10000000;
std::vector<double> y1, y2;

//Solve equation 1
y1.resize(N);
y1[0] = 0.;
for (unsigned int i=0; i<N-1; ++i){
  y1[i+1] = y1[i] + dt*(y1[i] + dt*i + 1);
}

//Solve equation 2
y2.resize(N);
y2[0] = 0.5;
for (unsigned int i=0; i<N-1; ++i){
  y2[i+1] = y2[i] + dt*(-y2[i] + (dt*i)*(dt*i));
}
```

- http://coliru.stacked-crooked.com/a/5d09526170c14e57

# Exercise 3: Possible solution

(...)

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    //Solve equation 1
    {
      y1.resize(N);
      y1[0] = 0.;
      for (unsigned int i=0; i<N-1; ++i){
        y1[i+1] = y1[i] + dt*(y1[i] + dt*i + 1);
      }
    }
    #pragma omp section
    //Solve equation 2
    {
      y2.resize(N);
      y2[0] = 0.5;
      for (unsigned int i=0; i<N-1; ++i){
        y2[i+1] = -y2[i] + dt*(y2[i] + (dt*i)*(dt*i));
      }
    }
  }
}
```

http://coliru.stacked-crooked.com/a/8087587fcdb1832e

# private and shared

- Variables declared outside a parallel block are shared by default (i.e. the value is shared by all threads).
- Variables declared inside a parallel block are private by default (i.e. the value can vary from thread to thread).
- Previously declared variables can be declared as private at the beginning of a parallel block (this may be useful when parallelizing code).
  **Serial**:

```
unsigned int i, j;
for (i=0; i<N; ++i){
  for (j=0; j<M; ++j){
    //do something;
  }
}
```

**Parallel**:

```
unsigned int i, j;
#pragma omp parallel for private(i,j)
for (i=0; i<N; ++i){
  for (j=0; j<M; ++j){
    //do something;
  }
}
```

# Exercise 4: `barrier` and `nowait`

- A `barrier` instructs all threads to "catch up" before moving on.
  - There is an implicit barrier at the end of a for loop
- `nowait` removes the implicit barrier after a for loop.
- **Where do you need a `barrier`, and where can you use `nowait` in the following example?** (modified from "omp-hands-on-SC08" tutorial):

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);
  //Need barrier or not?

#pragma omp for
  for(i=0;i<N;i++){
    C[i]=big_calc3(i,A);
  } //Implicit barrier - do you need it?

#pragma omp for
  for(i=0;i<N;i++){
    B[i]=big_calc2(C, i);
  } //Implicit barrier - do you need it?

  A[id] = big_calc4(id);
  //Need barrier or not?
}
```

# Exercise 4: barrier and nowait

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);
#pragma omp barrier //Wait for all threads (A is needed)

#pragma omp for
  for(i=0;i<N;i++){
    C[i]=big_calc3(i,A);
  } //Implicit barrier (C is needed)

#pragma omp for nowait
  for(i=0;i<N;i++){
    B[i]=big_calc2(C, i);
  } //Move on without waiting for all threads using nowait
    //(B is not needed)

  A[id] = big_calc4(id);
}
```

# collapse(j)

- `collapse` combines nested for loops into a single loop for parallelization.
- Operations must occur only within the inner loop.
- Effectiveness depends on the number of processors available.
  - Number of processors/cores available should be greater than the outer loop limit.
- The number of loops to be collapsed is specified.

```
#pragma omp parallel for collapse(2)
  for(i=0;i<M;i++){
    for(j=0;j<N;j++){
      function(i,j)
    }
  }
```

# Exercise 5: `collapse`

**For which of the following situations would `collapse` apply?**

1. With 8 cores available:

```
#pragma omp parallel for
  for(i=0;i<4;i++){
    b[i] = function1(i);
    for(j=0;j<4;j++){
      A[i][j]=b[i]*function2(i,j);
    }
  }
```

2. With 8 cores available:

```
#pragma omp parallel for
  for(i=0;i<16;i++){
    for(j=0;j<4;j++){
      A[i][j]=function3(i,j);
    }
  }
```

3. With 8 cores available:

```
#pragma omp parallel for
  for(i=0;i<4;i++){
    for(j=0;j<4;j++){
      A[i][j]=function3(i,j);
    }
  }
```

# Exercise 5: `collapse`

1. With 8 cores available: **No, not all operations are within the inner loop. Loops would need to be modified.**

```
#pragma omp parallel for
  for(i=0;i<4;i++){
    b[i] = function1(i);
    for(j=0;j<4;j++){
      A[i][j]=b[i]*function2(i,j);
    }
  }
```

2. With 8 cores available: **No, the outer loop goes up to 16, and we only have 8 cores. All cores will be already be used.**

```
#pragma omp parallel for
  for(i=0;i<16;i++){
    for(j=0;j<4;j++){
      A[i][j]=function3(i,j);
    }
  }
```

3. With 8 cores available: **Yes, we have 8 cores and an upper limit of 4 in the outer loop. Without `collapse`, 4 processors will be idle.**

```
#pragma omp parallel for collapse(2)
  for(i=0;i<4;i++){
    for(j=0;j<4;j++){
      A[i][j]=function3(i,j);
    }
  }
```

# Resources

- Slides available at github.com/greght/Workshop-OpenMP
- en.wikipedia.org/wiki/OpenMP
- www.openmp.org/resources/
- www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- Examples compiled, run, and saved using the online C++ compiler at coliru.stacked-crooked.com
- Free CSCAR consultations for UM student/faculty/staff research (cscar.research.umich.edu)