Advanced Computing for Engineering Applications

Dan Negrut
Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Darmstadt University
October 10, 2014
What We Covered Yesterday...

- GPU computing w/ thrust
- Parallel computing with OpenMP
  - Parallel regions
  - Work sharing
    - Parallel for
    - Parallel sections
    - Parallel tasks
  - Scheduling
  - Data scoping
Today…

- Parallel computing with OpenMP
  - Work sharing
    - Parallel tasks
  - Scheduling
  - Data scoping

- Parallel computing with the Message Passing Interface (MPI)
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing – Tasks
  - Data environment
  - Synchronization
OpenMP Tasks

- Task – Most important feature added as of OpenMP 3.0 version

- Allows parallelization of irregular problems
  - Unbounded loops (not clear how many iterations – see next example)
  - Recursive algorithms
  - Producer/consumer

- Start next with a motivating example
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= 14; i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```c
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
  if (i <= N)
    return N;
  else
    return 0;
}

int main() {
  int upperB = 14;

  for (int i = 0; i <= getUpperBound(i,upperB); i++){
    printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
  }

  printf("All done here...\n");
}
```

Code run on one thread, sequential execution, no OpenMP
```c
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N) {
    if (i <= N)
        return N;
    else
        return 0;
}

int main() {
    int upperB = 14;

    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= getUpperBound(i, upperB); i++) {
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```

```
1>------ Build started: Project: TestOpenMP, Configuration: Debug Win32 ------
1>   driverOpenMP.cpp
1>c:\users\negrut\bin\vs13projects\testopenmp\testopenmp\driveropenmp.cpp(15):
error C3017: termination test in OpenMP 'for' statement has improper form
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```
Tasks are independent units of work

A thread is assigned to perform a task

Tasks might be executed immediately or might be deferred
- The OS & runtime decide which of the above

Tasks are composed of
- **code** to execute
- **data** environment
- **internal control variables** (ICV)
Tasks: What Are They?
[MORE SPECIFICS…]

- **Code to execute**
  - The literal code in your program enclosed by the task directive

- **Data environment**
  - The shared & private data manipulated by the task

- **Internal control variables**
  - Thread scheduling and environment variable type controls

- A task is a specific instance of executable code and its data environment, generated when a thread encounters a `task` construct

- **Two activities:** (1) packaging, and (2) execution
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
```cpp
using namespace std;

typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.0;
}

int main() {
    LISTDBL test;  // default constructor
    LISTDBL::iterator it;

    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 8; ++j) test.insert(test.end(), pow(10.0, i + 1) + j);
    }

    for (it = test.begin(); it != test.end(); it++)
        cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        {
            while (it != test.end()) {
                #pragma omp task private(it)
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }

    for (it = test.begin(); it != test.end(); it++)
        cout << *it << endl;
    return 0;
}

#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>
```
Compile like:

```
$ g++ -o testOMP.exe testOMP.cpp
```
Task Construct – Explicit Task View

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – call this thread “L”
- Thread L runs the while loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task run by one thread
- All tasks complete at the barrier at the end of the parallel region’s construct
- Each task has its own stack space that will be destroyed when the task is completed
  - See example in a little bit

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p!=listEnd) { //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    {   // block 1
        node *p = head_of_list;
        while (p) { //block 2
            #pragma omp task private(p)
            {
                process(p);
            }
            p = p->next;    //block 3
        }
    }
}
Tasks: Synchronization Issues

● Setup:
  ● Assume Task B specifically relies on completion of Task A
  ● You need to be in a position to guarantee completion of Task A before invoking the execution of Task B

● Tasks are guaranteed to be complete at thread or task barriers:
  ● At the directive: `#pragma omp barrier`
  ● At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
#pragma omp task
foo();
#pragma omp barrier
#pragma omp single
{
#pragma omp task
bar();
}
}
```

- Multiple foo tasks created here - one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here

Task Completion Example:

- foo task guaranteed to be completed here
- One bar task created here
- Multiple foo tasks created here - one for each thread
**Comments: sections vs. tasks**

- **sections** have a "static" attribute: things are mostly settled at compile time

- The **tasks** construct is more recent and more sophisticated
  - They have a "dynamic" attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The run time system generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer

- **NOTE:** It’s the developer responsibility to ensure that different tasks can be executed concurrently; i.e., there is no data dependency
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization

- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- **shared** clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct
Data Scoping – What’s Private

- Not everything is shared...

  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- firstprivate
  - Specifies that each thread should have its own instance of a variable
  - Data is initialized using the value of the variable using the same name from the master thread
Example:
private vs. firstprivate

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

int main(void) {
    int i = 10;

    #pragma omp parallel private(i)
    {
        int threadID = omp_get_thread_num();
        printf("thread %d: i = %d\n", threadID, i);
        i = 1000 + threadID;
    }

    printf("i = %d\n", i);
    return 0;
}
```

Example:
private vs. firstprivate

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

int main(void) {
    int i = 10;

    #pragma omp parallel firstprivate(i)
    {
        int threadID = omp_get_thread_num();
        printf("threadID + i = \%d\n", threadID+i);
    }

    printf("i = \%d\n", i);

    return 0;
}

[stackoverflow]→
Other Tidbits

- There is a `lastprivate` flavor of private variable
  - The enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration of the work-sharing construct (`for`, `section`, `task`)
Data Scoping – The Basic Rule

- When in doubt, explicitly indicate who’s what
  - Data scoping: one of the most common sources of errors in OpenMP
```c
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }

    printf("Thread %d starting...\n",tid);

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++)
            {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }

        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++)
            {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    } /* end of sections */

    printf("Thread %d done.\n",tid);
} /* end of parallel section */
```

Q: Do you see any problem with this piece of code?

When in doubt, explicitly indicate who’s what.
A Data Environment Example

float A[10];
main () {
    int index[10];
#pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index) {
    float temp[10];
    static int count;
    <...
}
Assumed to be in another translation unit

A, index, and count are shared by all threads, but temp is local to each thread
Example: Data Scoping Issue - fib

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

int fib(int);

int main()
{
    int n = 10;
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
```
Example: Data Scoping Issue - \texttt{fib}

Assume that the parallel region exists outside of \texttt{fib} and that \texttt{fib} and the tasks inside it are in the dynamic extent of a parallel region.

\begin{verbatim}
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    \#pragma omp task
    x = fib(n-1);
    \#pragma omp task
    y = fib(n-2);
    \#pragma omp taskwait
    return x+y;
}
\end{verbatim}

Values of the private variables not available outside of tasks

$n$ is private in both tasks

$x$ is a private variable

$y$ is a private variable

This is very important here

What's wrong here?
Example: Data Scoping Issue - `fib`

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
#pragma omp task
    {
        x = fib(n-1);
    }
#pragma omp task
    {
        y = fib(n-2);
    }
#pragma omp taskwait
    return x+y
}
```

Values of the private variables not available outside of tasks

x is a private variable
y is a private variable
Example: Data Scoping Issue - `fib`

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait

    return x+y;
}
```

The values of the x & y variables will be available outside each task construct – after the taskwait.
```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = { 2, 4, 6 };
    int b[3] = { 1, 3, 5 };
    int c[3], d[3];
    int i, tid, nthreads;

    #pragma omp parallel private(i, tid) shared(a, b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n", tid);

        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Thread %d doing section 1\n", tid);
                for (i = 0; i < N; i++) {
                    c[i] = a[i] + b[i];
                    printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
                }
            }
            #pragma omp section
            {
                printf("Thread %d doing section 2\n", tid);
                for (i = 0; i < N; i++) {
                    d[i] = a[i] * b[i];
                    printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
                }
            }
        }
        /* end of sections */
        printf("Thread %d done.\n", tid);
    } /* end of parallel section */
    for (i = 0; i < N; i++) {
        printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
    }
    return 0;
}
```
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = {2, 4, 6};
    int b[3] = {1, 3, 5};
    int c[3], d[3];
    int i, tid, nthreads;

    #pragma omp parallel private(i,tid, c, d) shared(a,b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n", tid);

        #pragma omp section
        {
            printf("Thread %d doing section 1\n", tid);
            for (i = 0; i < N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
            }
        }

        #pragma omp section
        {
            printf("Thread %d doing section 2\n", tid);
            for (i = 0; i < N; i++) {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
            }
        }
        /* end of sections */
        printf("Thread %d done.\n", tid);
    } /* end of parallel section */
    for (i = 0; i < N; i++) {
        printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
    }
    return 0;
}
Work Plan

What is OpenMP?
- Parallel regions
- Work sharing
- Data environment

Synchronization

- Advanced topics
Implicit Barriers

- Several OpenMP constructs have *implicit* barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the **nowait** clause
  - The **nowait** clause is applicable to:
    - for clause
    - single clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{...};
```

```c
#pragma single nowait
{[...]}
```

```c
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
b[j] = bigFunc2(j);
```

Credit: IOMPP
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
#pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location

- Special case of a **critical** section, to be discussed shortly
  - Atomic introduces less overhead than **critical**

```c
#pragma omp parallel for shared(x, y, index)
for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

Credit: IOMPP
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is wrong here?

Credit: IOMPP
Race Condition

- A *race condition* - two or more threads access a shared variable at the same time.
  - Leads to nondeterministic behavior

- For example, suppose that *area* is shared and both Thread A and Thread B are executing the statement

\[
\text{area } += 4.0 / (1.0 + x* x);
\]
Two Possible Scenarios

<table>
<thead>
<tr>
<th>Value of area</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td>+3.765</td>
<td>+3.563</td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.995</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order of thread execution causes non-determinant behavior in a data race.

Credit: IOMPP
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++)
    {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```c
float RES;
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<niters; i++) {
        float B = big_job(i);
        #pragma omp critical (RES_lock)
        consum(B, RES);
    }
}
```

- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional but highly recommended.

Includes material from IOMPP
reduction Example

```c
#pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
```

- Local copy of `sum` for each thread engaged in the reduction is private
  - Each local sum initialized to the identity operand associated with the operator that comes into play
    - Here we have “+”, so it’s a zero (0)

- All local copies of `sum` added together and stored in “global” variable

Credit: IOMPP
OpenMP reduction Clause

reduction (op:list)

- The variables in list will be shared in the enclosing parallel region

- Here’s what happens inside the parallel or work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads

- At end of construct, local copies are combined through “op” into a single value
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

```c
static long num_steps=100000;
double step, pi;

void main() {
    int i;
double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Credit: IOMPP
```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

#pragma omp parallel for reduction(+:sum)
{
    for(int i=0; i<num_steps; i++) {
        double x = (i + .5)*step;
        sum += 4.0/(1.+ x*x);
    }
}

double my_pi = sum*step;
printf("Value of integral is: \%f\n", my_pi);

return 0;
}
```
OpenMP Reduction Example:

Output

[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o test.exe
[negrut@euler24 CodeBits]$ ./test.exe 100000
Value of integral is: 3.141593
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Credit: IOMPP
Example: Variable Scoping Aspects

- Consider parallelizing the following code

```c
int main() {
    const int n=20;
    int a[n];
    for( int i=0; i<n; i++ )
        a[i] = i;

    //this is the part that needs to
    //be parallelized
    caller(a, n);

    for( int i=0; i<n; i++ )
        printf("a[%d]=%d\n", i, a[i]);
    return 0;
}

void caller(int *a, int n) {
    int i, j, m=3;
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}

void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<n; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the “callee” subroutine

- If you run the executable 20 times, you get the same results 20 times
First Attempt to Parallelize

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad…
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 70
    ```

- If you run executable 20 times, you get different results

- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)
  - Fortunately, it’s not used in this code for any subsequent computation

- Q: How would you fix this issue with the static variable?
  - Not necessarily to print the values in increasing order, but to make sure there are no race conditions
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)
Program Output, Parallelized Code

- It looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely
  - No reason for this behavior to change
Concluding Remarks on the OpenMP API
OpenMP: 30,000 Feet Perspective

- Good momentum behind OpenMP owing to the ubiquity of the multi-core chips
- Shared memory, thread-based parallelism
- Relies on the programmer defining parallel regions
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
  - OpenMP 4.0 – complete specifications released in July 2013
OpenMP
The 30,000 Feet Perspective

- Nomenclature:
  - Multicore Communication API (MCAPI)
  - Multicore Resource-sharing API (MRAPI)
  - Multicore Task Management API (MTAPI)
The OpenMP API

- The OpenMP API is a combination of
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`
The “directives” fall into three categories:

- **Expression of parallelism (flow control)**
  - Example: `#pragma omp parallel for`

- **Data sharing among threads (communication)**
  - Example: `#pragma omp parallel for private(x,y)`

- **Synchronization (coordination or interaction)**
  - Example: `#pragma omp barrier`
OpenMP 4.0:
Subset of Run-Time Library OpenMP Routines

1. `omp_set_num_threads`
2. `omp_get_num_threads`
3. `omp_get_max_threads`
4. `omp_get_thread_num`
5. `omp_get_thread_limit`
6. `omp_get_num_procs`
7. `omp_in_parallel`
8. `omp_set_dynamic`
9. `omp_get_dynamic`
10. `omp_set_nested`
11. `omp_get_nested`
12. `omp_set_schedule`
13. `omp_get_schedule`
14. `omp_set_max_active_levels`
15. `omp_get_max_active_levels`
16. `omp_get_level`
17. `omp_get_ancestor_thread_num`
18. `omp_get_team_size`
19. `omp_get_active_level`
20. `omp_init_lock`
21. `omp_destroy_lock`
22. `omp_set_lock`
23. `omp_unset_lock`
24. `omp_test_lock`
25. `omp_init_nest_lock`
26. `omp_destroy_nest_lock`
27. `omp_set_nest_lock`
28. `omp_unset_nest_lock`
29. `omp_test_nest_lock`
30. `omp_get_wtime`
31. `omp_get_wtick`
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables
[select env variables]

- **OMP_STACKSIZE**
  - Controls the size of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides a hint to an OpenMP implementation about the desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program Example:
    - `setenv OMP_THREAD_LIMIT 8`
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can implement complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
OpenMP, Some Caveats

- There is a lag between the moment a new specification is released and the time a compiler is capable of handling all of its aspects
  - Intel’s compiler is probably most up to speed

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
- OpenMP.org, [http://openmp.org/](http://openmp.org/)
- OpenMP 3.0 API Summary Cards:
  - C/C++: [http://openmp.org/mp-documents/OpenMP-4.0-C.pdf](http://openmp.org/mp-documents/OpenMP-4.0-C.pdf)
- [http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf](http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
Parallel Computing with MPI
Acknowledgments

- Parts of MPI material covered draws on a set of slides made available by the Irish Centre for High-End Computing (ICHEC) - [www.ichec.ie](http://www.ichec.ie)
  - These slides will contain “ICHEC” at the bottom
  - In turn, the ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh

- Individual or institutions are acknowledged at the bottom of the slide, like [A. Jacobs]→
MPI: Textbooks, Further Reading…

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 - very good introduction.
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC
Shared Memory Systems

- Memory resources are shared among processors
  - Typical scenario, on a budget: one node with four CPUs, each with 16 cores
  - Typically one node systems

- Two issues:
  - Scales poorly with system size due to the need for cache coherence
  - Most often, you need more memory than available on the typical multi-core node

- Example:
  - Symmetric Multi-Processors (SMP)
    - Each processor has equal access to RAM

- Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing
Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS

- Information is passed between nodes using the network

- No cache coherence and no need for special cache coherency hardware

- Software development: more difficult to write programs for distributed memory systems since the programmer must keep track of memory usage

- Traditionally, this represents the hardware setup for big-iron computing
Overview of Large Multiprocessor Hardware Configurations

- Larger multiprocessors
  - Shared address space
    - Symmetric shared memory (SMP)
      - Examples: IBM eserver, SUN Sunfire
    - Distributed shared memory (DSM)
  - Distributed address space
    - Commodity clusters: Beowulf and others
    - Custom cluster
      - Cache coherent: ccNUMA
        - SGI Origin/Altix
      - Noncache coherent: Cray T3E, X1
      - Euler
      - Uniform cluster: IBM BlueGene
      - Constellation cluster of DSMs or SMPs
        - SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Euler
~ Hardware Configurations ~

Legend, Connection Type:
- Gigabit Ethernet
- 4x QDR Infiniband

File Server Architecture:
- CPU: Intel Xeon 5620
- RAM: 16 GB DDR3
- InfiniBand HCA
- RAID 6
- 24x 2TB Hard Disks

Remote Collaborators

Gigabit Ethernet Switch

Head Node

4x QDR Infiniband Switch

File Server
CPU/GPU Node 1
CPU/GPU Node 2
CPU/GPU Node 14
AMD Node 1

CPU/GPU Node Architecture:
- CPU 0: Intel Xeon 5520
- CPU 1: Intel Xeon 5520
- RAM: 48 GB DDR3
- GPU 0
- GPU 1
- GPU 2
- GPU 3
- 64 GB RAM
- 1.508 GB RAM
- 448 Cores
- PCIe x16 2.0

AMD Node Architecture:
- CPU 0: AMD Opteron 6276
- CPU 1: AMD Opteron 6276
- RAM: 128 GB DDR3
- InfiniBand HCA
- SSD

Internet

Internal Users
Hardware Relevant in the **Context of MPI**

Two Components of Euler that are Important

- **CPU**: AMD Opteron 6274 Interlagos 2.2GHz
  - 16-Core Processor (four CPUs per node → 64 cores/node)
  - 8 x 2MB L2 Cache per CPU
  - 2 x 8MB L3 Cache per CPU
  - Thermal Design Power (TDP): 115W

- **HCA**: 40Gbps Mellanox Infiniband interconnect
  - Bandwidth comparable to PCIe2.0 x16 (~32Gbps), yet the latency is rather poor (~1microsecond)
  - Ends up being the bottleneck in cluster computing
The same program is launched for execution independently on a collection of cores.

Each core executes the program.

What differentiates processes is their rank: processes with different ranks do different things ("branching based on the process rank")

- Very similar to GPU computing, where one thread did work based on its thread index
- Very similar to OpenMP function omp_get_thread_num()
The Message-Passing Model

- One starts many processes on different cores but on each core the process is spawned by launching the same program
  - Each process has its own program counter and address space

- Message passing enables communication among processes that have separate address spaces

- Interprocess communication typically consists of
  - Synchronization, followed by…
  - … movement of data from one process’s address space to another’s

- Execution paradigm embraced in MPI: Single Program Multiple Data (SPMD)
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**
  - data
  - memory
  - Processor/Process

- **Message-Passing Programming Paradigm**
  - data
  - Distributed memory
  - Parallel processors
  - communication network
A **process** is a **software program** executing a task on a **processor**

In other words, each processor/process in a message passing parallel job runs an instance/copy of a **program**:

- Program written in a conventional sequential language, e.g., C or Fortran

- The variables of each program have the same name but different locations (distributed memory) and different data

- Communicate via special send & receive routines (**message passing**)

![Diagram of communication network and processes]
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    }
    else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize();
    return 0;
}
Program Output

```
[negrut@euler04 CodeBits]$ mpiexec -np 8  ./a.out
I am a worker: euler04 (rank=1/7)
I am a worker: euler04 (rank=5/7)
I am a worker: euler04 (rank=6/7)
I am a worker: euler04 (rank=3/7)
I am a worker: euler04 (rank=4/7)
I am the master: euler04
I am a worker: euler04 (rank=2/7)
I am a worker: euler04 (rank=7/7)
[negrut@euler04 CodeBits]$
[negrut@euler04 CodeBits]$
```
Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - Some of these supercomputers might use GPU acceleration though
  - The program executed can also use OpenMP on a node

- Examples of architectures relying on MPI for HPC:
  - IBM Blue Gene L/P/Q (Argonne National Lab – “Mira”)
  - Cray supercomputers (Oakridge National Lab – “Titan”, also uses K20X GPUs)

- MPI has FORTRAN, C, and C++ bindings – widely used in Scientific Computing
MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- One of the early common implementations: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - OpenMPI: joint effort of three or four groups (Los Alamos, Tennessee, Indiana University, Europe)
Where Can We Use Message Passing?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of Workstations
  - Even on shared memory systems
MPI vs. CUDA

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?
  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems

- In large scale computing the future likely to belong to heterogeneous architecture
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)
Parallel computing on the GPU, on multi-core chips, and multi-node (MPI) share one thing:

- Same code is executed but for different data
- There is some sort of ID that helps a thread/process figure out what data the thread/process needs to work on
  - In CUDA you have threadIdx and blockIdx
  - In OpenMP you have the thread ID
  - In MPI you have a process rank

There are exceptions from the rule (thread divergence, sections, etc.)
MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, newer edition available

```c
/* greetings.c -- greetings program
 * Send a message from all processes with rank != 0 to process 0.
 * Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank;  /* rank of process */
    int p;       /* number of processes */
    int source;  /* rank of sender */
    int dest;    /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that \0 gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
MPI, a Third Example: Approximating $\pi$

\[
\int_0^1 \frac{4}{1 + x^2} = 4 \cdot \tan^{-1}(1) = \pi
\]

Numerical Integration: Midpoint rule

\[
\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f \left( (i - 0.5) \cdot h \right)
\]
MPI, a Third Example: Approximating $\pi$

- Use 4 MPI processes (rank 0 through 3)
- In the picture, $n=13$
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: 1, 5, 9, 13
  - Rank 1: 2, 6, 10
  - Rank 2: 3, 7, 11
  - Rank 3: 4, 8, 12
- Each rank computes the area in its associated sub-intervals
- **MPIReduce** is used to sum the areas computed by each rank yielding final approximation to $\pi$
```
// MPI_PI.cpp : Defines the entry point for the console application.
//
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
```
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) ";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
    cout << "pi is approximately " << pi <<", Error is " << fabs(pi - PI25DT) << endl;
}

MPI_Finalize();
return 0;
Broadcast

[MPI function used in Example]

- A one-to-many communication.
Collective Communications

- Collective communication routines are higher level routines
- Several processes are involved at a time
- May allow optimized internal implementations, e.g., tree based algorithms
  - Require $O(\log(N))$ time as opposed to $O(N)$ for naïve implementation
Reduction Operations

[MPI function used in Example]

- Combine data from several processes to produce a single result
Barriers

- Used implicitly or explicitly to synchronize processes
MPI, Practicalities
MPI on Euler

[Selecting MPI Distribution]

- What’s available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler

- To load OpenMPI environment variables:
  - (This should have been done automatically)

$ module load mpi/gcc/openmpi
MPI on Euler:
[Compiling MPI Code by Hand]

- Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
  - Adds in `-L`, `-l`, `-I`, etc. flags for MPI
  - Passes any options to your native compiler (`gcc`)
  - Very similar to what `nvcc` did for CUDA – it’s a compile driver...

```
$ mpicxx -o integrate_mpi integrate_mpi.cpp
```
Running MPI Code on Euler

\texttt{mpiexec [-np #] [-machinefile file] <program> [<args>]}

- The machinefile/nodefile is required for multi-node jobs with the version of OpenMPI on Euler
- \texttt{-np} will be set automatically from the machinefile; can select lower, but not higher
- See the \texttt{mpiexec} manpage for more options
Example

euler $ qsub -I -l nodes=8:ppn=4:amd,walltime=5:00
qsub: waiting for job 15246.euler to start
qsub: job 15246.euler ready

euler07 $ cd $PBS_O_WORKDIR

Code

euler07 $ mpiexec -machinefile $PBS_NODEFILE ./integrate_mpi
32 32.121040666358297 in 0.998202s

euler07 $ mpiexec -np 16 -machinefile $PBS_NODEFILE ./integrate_mpi
16 32.121040666359455 in 1.524001s

euler07 $ mpiexec -np 8 -machinefile $PBS_NODEFILE ./integrate_mpi
8 32.121040666359136 in 2.171963s

euler07 $ mpiexec -np 4 -machinefile $PBS_NODEFILE ./integrate_mpi
4 32.121040666360585 in 4.600204s

euler07 $ mpiexec -np 2 -machinefile $PBS_NODEFILE ./integrate_mpi
2 32.121040666367888 in 7.615060s

euler07 $ ./integrate_mpi
1 32.121040666353437 in 15.163330s
MPI Nuts and Bolts
Goals/Philosophy of MPI

- MPI’s prime goals
  - Provide a message-passing interface for parallel computing
  - Make source-code portability a reality
  - Provide a set of services (building blocks) that increase developer’s productivity

- The philosophy behind MPI:
  - Specify a standard and give vendors the freedom to go about its implementation
  - Standard should be hardware platform & OS agnostic – key for code portability
The Rank, as a Facilitator for Data and Work Distribution

- To communicate together MPI processes need identifiers: \( \text{rank} = \text{identifying number} \)

- Work distribution decisions are based on the \( \text{rank} \)
  - Helps establish which process works on which data
  - Just like we had thread and block indices in CUDA

\[ \text{myrank} = 0 \quad \text{data} \]
\[ \text{program} \]

\[ \text{myrank} = 1 \quad \text{data} \]
\[ \text{program} \]

\[ \text{myrank} = 2 \quad \text{data} \]
\[ \text{program} \]

\[ \text{myrank} = (\text{size}-1) \quad \text{data} \]
\[ \text{program} \]
Message Passing

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:
  - sending process + receiving process \{ i.e., the two “ranks” \}
  - source location + destination location
  - source data type + destination data type
  - source data size + destination buffer size

[Diagram of message passing through a communication network]
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
Communicator  MPI_COMM_WORLD

- All processes of an MPI program are members of the default communicator MPI_COMM_WORLD

- MPI_COMM_WORLD is a predefined handle in mpi.h

- Each process has its own rank in a given communicator:
  - starting with 0
  - ending with (size-1)

- You can define a new communicator in case you find it useful
  - Use MPI_Comm_create call. Example creates the communicator DANS_COMM_WORLD

    ```
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &DANS_COMM_WORLD);
    ```
**MPI_Comm_create**

- **Synopsis**
  
  ```c
  int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
  ```

- **Input Parameters**
  - `comm` - communicator (handle)
  - `group` - subset of the family of processes making up the `comm` (handle)

- **Output Parameter**
  - `comm_out` - new communicator (handle)
Point-to-Point Communication

- Simplest form of message passing

- One process sends a message to another process
  - MPI_Send
  - MPI_Recv

- Sends and receives can be
  - Blocking
  - Non-blocking
  - More on this shortly
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., `DANS_COMM_WORLD`
- Processes are identified by their ranks in the communicator

![Diagram showing point-to-point communication with processes labeled 0 to 6 and a message from source to destination within a communicator](image-url)
The Data Type

- A message contains a number of elements of some particular data type

- MPI data types:
  - Basic data type
  - Derived data types – more on this later

- Data type handles are used to describe the type of the data moved around

Example: message with 5 integers

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>654</td>
<td>96574</td>
<td>-12</td>
<td>7676</td>
<td></td>
</tr>
<tr>
<td>MPI Datatype</td>
<td>C datatype</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_LONGDOUBLE</td>
<td>long double</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

```plaintext
count=5
int arr[5]
datatype=MPI_INT
```

2345  654  96574  -12  7676
MPI_Send & MPI_Recv: The Eager and Rendezvous Flavors

- If you send small messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “eager mode”

- If you send a large amount of data, the sender function waits for the receiver to post a receive before sending the actual data of the message

- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send a small amount of data, the MPI implementation can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play this trick if you attempt to move around a huge chunk of data though
NOTE: Each implementation of MPI has a default value (which might change at run time) beyond which a larger `MPI_Send` stops acting “eager”
  - The MPI standard doesn’t provide specifics
  - You don’t know how large is too large…

Does it matter if it’s Eager or Rendezvous?
  - In fact it does, sometimes the code can hang – example to come

Remark: In the message-passing paradigm for parallel programming you’ll always have to deal with the fact that the data that you send needs to “live” somewhere during the send-receive transaction
MPI_Send & MPI_Recv: Blocking vs. Non-blocking

- Moving away from the Eager vs. Rendezvous modes → they only concern the MPI_Send and MPI_Recv pair

- Messages can be sent with other vehicles than plain vanilla MPI_Send

- The class of send-receive operations can be classified based on whether they are blocking or non-blocking
  - Blocking send: upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since a copy of the data has been sent
  - Non-blocking: the send call returns immediately and there is no guarantee that the data has actually been transmitted upon return from send call
    - Take home message: before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed
Example: Send & Receive

Non-blocking Alternative: MPI_Isend

- If non-blocking, the data “lives” in your buffer – that’s why it’s not safe to change it since you don’t know when transaction was closed
  - This typically realized through a MPI_Isend
    - “I” stands for “immediate”

- NOTE: there is another way for providing a buffer region but this alternative is blocking
  - Realized through MPI_Bsend
    - “B” stands for “buffered”
  - The problem here is that *you* need to provide this additional buffer that stages the transfer
    - Interesting question: how large should *that* staging buffer be?
  - Adding another twist to the story: if you keep posting non-blocking sends that are not matched by corresponding “MPI_Recv” operations, you are going to overflow this staging buffer
Example: Send & Receive

Blocking Options (several of them)

- The plain vanilla MPI_Send & MPI_Recieve pair is blocking
  - It’s safe to modify the data buffer upon return

- The problem with plain vanilla:
  - 1: when sending large messages, there is no overlap of compute & data movement
    - This is what we strived for when using “streams” in CUDA
  - 2: if not done properly, the processes executing the MPI code can hang

- There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above
The Mechanics of P2P Communication: Sending a Message

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- **buf** is the starting point of the message with **count** elements, each described with **datatype**

- **dest** is the rank of the destination process within the communicator **comm**

- **tag** is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The **tag** can be used to distinguish between different messages
  - Rarely used
The Mechanics of P2P Communication: Receiving a Message

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

- **buf/count/datatype** describe the receive buffer
- Receiving the message sent by process with rank **source** in **comm**
- Only messages with matching **tag** are received
- Envelope information is returned in the **MPI_Status** object **status**
 MPI_Recv:  
The Need for an MPI_Status Argument

- The MPI_Status object returned by the call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound
  - If multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation
The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – source = MPI_ANY_SOURCE
  - To receive from any tag – tag = MPI_ANY_TAG
  - Actual source and tag returned in receiver's status argument
The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI_RECV in status.

- `status.MPI_SOURCE`
- `status.MPI_TAG`
- `count` via `MPI_Get_count()`

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
```

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message data types must match
- Receiver’s buffer must be large enough
Blocking Type: Communication Modes

- Send communication modes:
  - Synchronous send \(\rightarrow\) MPI_SSEND
  - Buffered [asynchronous] send \(\rightarrow\) MPI_BSEND
  - Standard send \(\rightarrow\) MPI_SEND
  - Ready send \(\rightarrow\) MPI_RSEND

- Receiving all modes \(\rightarrow\) MPI_RECV
# Cheat Sheet, Blocking Options

<table>
<thead>
<tr>
<th>Sender modes</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_SSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td>needs application-defined buffer to be declared with <strong>MPI_BUFFER_ATTACHMENT</strong></td>
</tr>
<tr>
<td><strong>MPI_BSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classic</td>
<td>Standard send</td>
<td>Rendezvous or eager mode. Decided at run time</td>
</tr>
<tr>
<td><strong>MPI_SEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>Started right away. Will work out <strong>only</strong> if the matching receive is already posted!</td>
<td>Blindly do a send. Avoid, might cause unforeseen problems...</td>
</tr>
<tr>
<td><strong>MPI_RSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message (data) has arrived</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_RECV</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1) Synchronous Sending in MPI
2) Buffered Sending in MPI

- **Synchronous with MPI_Ssend**
  - In synchronous mode, a send will not complete until a matching receive is posted.
  - The sender has to wait for a receive to be posted
  - No buffering of data
  - Used for ensuring the code is healthy and doesn’t rely on buffering

- **Buffered with MPI_Bsend**
  - Send completes once message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsend and no matching receives have been posted yet
3) Standard Sending in MPI
4) Ready Sending in MPI

- **Standard with MPI_Send**
  - Up to the MPI implementation to decide whether to do rendezvous or eager, for performance reasons
    - NOTE: If it does rendezvous, in fact the behavior is that of MPI_SSend
  - Very commonly used

- **Ready with MPI_Rsend**
  - Will work correctly *only* if the matching receive has been posted
  - Can be used to avoid handshake overhead when program is known to meet this condition
  - Rarely used, can cause major problems
Most Important Issue: Deadlocking

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

---

Process 0

... 
MPI_Ssend() 
MPI_Recv() 
...
...
MPI_Buffer_attach() 
MPI_Bsend() 
MPI_Recv() 
...
...
MPI_Buffer_attach() 
MPI_Bsend() 
MPI_Recv() 
...
...

Deadlock

No Deadlock

Process 1

... 
MPI_Ssend() 
MPI_Recv() 
...
...
MPI_Buffer_attach() 
MPI_Bsend() 
MPI_Recv() 
...
...

No Deadlock

No Deadlock
Deadlocking, Another Example

- MPI_Send can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:

```
Process 0
...
MPI_Send()
MPI_Recv()
...
```

```
Process 1
...
MPI_Send()
MPI_Recv()
...
```

Deadlock
Data size > 127999 bytes
Data size < 128000 bytes

No Deadlock
Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair `MPI_Ssend` and `MPI_Recv` operations the right way:

  ```plaintext
  Process 0
  ...
  MPI_Ssend()
  MPI_Recv()
  ...

  Process 1
  ...
  MPI_Recv()
  MPI_Ssend()
  ...

  No Deadlock
  ```

- Conclusion: understand how the implementation works and what its pitfalls/limitations are
Example

- Always succeeds, even if no buffering is done

```c
if (rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
elif (rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Will always deadlock, no matter the buffering mode

```c
if(rank==0)
{
    MPI_Recv(...);
    MPI_Send(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Only succeeds if message is at least one of the transactions is small enough and an “eager” mode is triggered

```c
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Send(...);
    MPI_Recv(...);
}
```
Concluding Remarks, Blocking Options

- Standard send (**MPI_SEND**)  
  - minimal transfer time  
  - may block due to synchronous mode  
  - \(\rightarrow\) risks with synchronous send

- Synchronous send (**MPI_SSEND**)  
  - risk of deadlock  
  - risk of serialization  
  - risk of waiting \(\rightarrow\) idle time  
  - high latency \(\rightarrow\) best bandwidth

- Buffered send (**MPI_BSEND**)  
  - low latency \(\rightarrow\) bad bandwidth

- Ready send (**MPI_RSEND**)  
  - use **never**, except you have a 200% guarantee thatRecv is already called in the current version and all future versions of your code
Technicalities, Loose Ends: More on the Buffered Send

- Relies on the existence of a buffer, which is set up through a call:
  ```c
  int MPI_Buffer_attach(void* buffer, int size);
  ```

- A bsend is a **local** operation. It does not depend on the occurrence of a matching receive in order to complete.

- If a bsend operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete.

- Return from an **MPI_Bsend** does not guarantee the message was sent.

- Message may remain in the buffer until a matching receive is posted.
Technicalities, Loose Ends: More on the Buffered Send [Cntd.]

- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space.

- The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient.

- **MPI_Bsend** lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data.

- The **MPI_Buffer_attach** subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a process at any time.
Technicalities, Loose Ends: Message Order Preservation

- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
  - Messages do not overtake each other
  - True even for non-synchronous sends

- If both receives match both messages, then the order is preserved
Timing an MPI Job

- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`

```c
int main()
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ...
    endtime = MPI_Wtime();
    printf("That took %f seconds\n", endtime - starttime);
    return 0;
}
```
Non-Blocking Communication
Non-Blocking Communications: Motivation

- Overlap communication with execution (just like with CUDA):
  - Initiate non-blocking communication
    - Returns immediately
    - Routine name starting with MPI_...
  - Do some work
    - “latency hiding”
  - Wait for non-blocking communication to complete
Non-blocking Send/Receive

- Syntax

```c
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

- buf - [in] initial address of send buffer (choice)
- count - [in] number of elements in send buffer (integer)
- datatype - [in] datatype of each send buffer element (handle)
- dest - [in] rank of destination (integer)
- tag - [in] message tag (integer)
- comm - [in] communicator (handle)
- request - [out] communication request (handle)

```c
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);
```
The Screenplay: Non-Blocking P2P Communication

- Non-blocking send

  ```
  MPI_Isend(...)  
  doing some other work  
  MPI_Wait(...)  
  ```

  = waiting until operation locally completed

- Non-blocking receive

  ```
  MPI_Irecv(...)  
  doing some other work  
  MPI_Wait(...)  
  ```

  = waiting until operation locally completed
Non-Blocking Send/Receive
Some Tools of the Trade

- Call returns immediately. Therefore, user must worry whether …
  - Data to be sent is out of the send buffer before trampling on the buffer
  - Data to be received has finished arriving before using the content of the buffer

- Tools that come in handy:
  - For sends and receives in flight
    - `MPI_Wait` – blocking - you go synchronous
    - `MPI_Test` – non-blocking - returns quickly with status information
  - Check for existence of data to receive
    - Blocking: `MPI_Probe`
    - Non-blocking: `MPI_Iprobe`
Waiting for isend/ireceive to Complete

- Waiting on a single send
  
  ```c
  int MPI_Wait(MPI_Request *request, MPI_Status *status);
  ```

- Waiting on multiple sends (get status of all)
  - Till all complete, as a barrier
    ```c
    int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
    ```
  - Till at least one completes
    ```c
    int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status);
    ```
  - Helps manage progressive completions
    ```c
    int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount,
                     int *indices, MPI_Status *statuses);
    ```
MPI_Test...

- Flag true means completed

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status);
```

- Like a non blocking MPI_Waitsome

```c
int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses);
```
The Need for MPI_Probe and MPI_Iprobe

- The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them.

- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status).

- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.
Probe to Receive

- Probes yield incoming size

- Blocking Probe, wait till match
  
  ```c
  int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
  ```

- Non Blocking Probe, flag true if ready
  
  ```c
  int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
  ```
Two types of communication:

- Blocking:
  - Safe to change content of buffer holding on to data in the MPI send call
- Non-blocking:
  - Be careful with the data in the buffer, since you might step on/use it too soon

MPI provides four modes for these two types

- standard, synchronous, buffered, ready
Collective Actions
Collective Actions

- MPI actions involving a group of processes
- Must be called by all processes in a communicator
- All collective actions are blocking

Types of Collective Actions (three of them):
- Global Synchronization (barrier synchronization)
- Global Communication (broadcast, scatter, gather, etc.)
- Global Operations (sum, global maximum, etc.)
Barrier Synchronization

- Syntax:

```c
int MPI_Barrier(MPI_Comm comm);
```

- **MPI_Barrier** not needed that often:
  - All synchronization is done automatically by the data communication
    - A process cannot continue before it has the data that it needs
  - If used for debugging
    - Remember to remove for production release
Communication Action: Broadcast

- Function prototype:

```c
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

- rank of the sending process (i.e., root process)
- must be given identically by all processes
MPI_Bcast

\[ A_0 \]: any chunk of contiguous data described with MPI_Datatype and count
MPI_Bcast

```c
int MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
```

- **INOUT**: `buffer` (starting address, as usual)
- **IN**: `count` (number of entries in buffer)
- **IN**: `type` (can be user-defined)
- **IN**: `root` (rank of broadcast root)
- **IN**: `com` (communicator)

- Broadcasts message from `root` to all processes (including `root`)
- `com` and `root` must be identical on all processes
- On return, contents of `buffer` is copied to all processes in `com`
Example: MPI_Bcast

- Read a parameter file on a single processor and send data to all processes

```c
#include "mpi.h"
#include <assert.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs;
    float data = -1.0;
    FILE *file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if( myRank==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file != NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);

    MPI_Finalize();
}
```
Example: MPI_Bcast

[Output]

```
[negrut@euler CodeBits] $ qsub -I -l nodes=8:ppn=4,walltime=5:00
qsub: waiting for job 16114.euler to start
qsub: job 16114.euler ready

[negrut@euler17 CodeBits] $ mpicxx testMPI.cpp
[negrut@euler17 CodeBits] $ mpiexec -np 4 a.out
data before: -1.000000
data before: -1.000000
data before: -1.000000
data after: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
```
Communication Action: Gather

- Function Prototype

```c
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

*e.g., root=1*

![Diagram of Gather function](chart.png)
MPI_Gather

[A. Siegel]→

processes

A₀
A₁
A₂
A₃
A₄
A₅

data

Gather

processes

A₀  A₁  A₂  A₃  A₄  A₅

A₁
A₂
A₃
A₄
A₅

data (buffer)
MPI_Gather

```c
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** `sendbuf` (starting address of send buffer)
- **IN** `sendcount` (number of elements in send buffer)
- **IN** `sendtype` (type)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `recvcount` (n-elements for any single receive)
- **IN** `recvtype` (data type of recv buffer elements)
- **IN** `root` (rank of receiving process)
- **IN** `comm` (communicator)
MPI_Gather

- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- Remarks:
  - Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
  - `recvcount` on root indicates number of items received from each process, not total. This is a very common error
- Exercise: Sketch an implementation of `MPI_Gather` using only send and receive operations.
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv) {
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = nlcl */
    data_loc = (float *) malloc(nlcl*sizeof(float));

    for (i = 0; i < nlcl; ++i) data_loc[i] = myRank;

    if (myRank == 0) data = (float *) malloc(nprocs*sizeof(float)*nlcl);

    MPI_Gather(data_loc, nlcl, MPI_FLOAT, data, nlcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0){
        for (i = 0; i < nlcl*nprocs; ++i){
            printf("%f\n", data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}

[A. Siegel]→
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
Communication Action: Scatter

- Function prototype

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

e.g., root=1
MPI_Scatter

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
</table>

Data (buffer) to processes:

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
</table>

Scatter:

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
</table>

Data (buffer) from processes:

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
</table>

[A. Siegel]→
### MPI_Scatter

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** `sendbuf` (starting address of send buffer)
- **IN** `sendcount` (number of elements **sent to each process**)
- **IN** `sendtype` (type)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `recvcount` (n-elements **in receive buffer**)
- **IN** `recvtype` (data type of receive elements)
- **IN** `root` (rank of sending process)
- **IN** `comm` (communicator)
MPI_Scatter

- Inverse of MPI_Gather

- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter

- Remarks:
  - All arguments are significant on root, while on other processes only recvbuf, recvcount, recvtype, root, and comm are significant
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv)
{
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ){
            for (int j = 0; j < n_lcl; ++j) printf("%f\n", data_l[j]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}

This is interesting.
Think what’s happening here...

[A. Siegel]→ 169
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$
Putting Things in Perspective...

- **Gather**: you automatically create a serial array from a distributed one

- **Scatter**: you automatically create a distributed array from a serial one
Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- \( d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1} \)
  - \( d_i = \) data in process rank \( i \)
    - single variable, or
    - vector
  - \( \circ = \) associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

- Floating point rounding may depend on usage of associative law:
  - \( ((d_0 \circ d_1) \circ (d_2 \circ d_3)) \circ \ldots \circ (d_{s-2} \circ d_{s-1}) \)
  - ((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \ldots ) \circ d_{s-2}) \circ d_{s-1})}
Example of Global Reduction

- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`.
- Assume `root=0`;

```c
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

- The result is only placed in `resultbuf` at the root process.
## Predefined Reduction Operation Handles

<table>
<thead>
<tr>
<th>Predefined operation handle</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of the maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of the minimum</td>
</tr>
</tbody>
</table>
MPI_Reduce

before MPI_REDUCE

• inbuf
  • result

after

root=1

AoDoGoJoM

[ICHEC]
Reduce Operation

Assumption: Rank 0 is the root
int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
MPI_Reduce example

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, 6, \text{MPI\_INT}, \text{MPI\_SUM}, 0, \text{MPI\_COMM\_WORLD})
\]

\[
\begin{array}{cccccc}
\text{P0} & 3 & 4 & 2 & 8 & 12 & 1 \\
\text{P1} & 5 & 2 & 5 & 1 & 7 & 11 \\
\text{P2} & 2 & 4 & 4 & 10 & 4 & 5 \\
\text{P3} & 1 & 6 & 9 & 3 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{rbuf} & 11 & 16 & 20 & 22 & 24 & 18 \\
\end{array}
\]

sbuf

P0

\[
\begin{array}{cccccc}
\text{rbuf} & 11 & 16 & 20 & 22 & 24 & 18 \\
\end{array}
\]
**MPI_Reduce, MPI_Allreduce**

- **MPI_Reduce**: result is collected by the root only
  - The operation is applied element-wise for each element of the input arrays on each processor

- **MPI_Allreduce**: result is sent out to everyone

```
MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
```

```
MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD)
```

Credit: Allan Snavely
MPI_Allreduce

<table>
<thead>
<tr>
<th>processes</th>
<th>data (buffer)</th>
<th>Allreduce</th>
<th>data (buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A0</td>
<td>B0</td>
<td>C0</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td></td>
</tr>
</tbody>
</table>
MPI_Allreduce

```c
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `comm` (communicator)
Example: MPI_Allreduce

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int my_rank, nprocs, gsum, gmax, gmin, data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    data_l = my_rank;

    MPI_Allreduce(&data_l, &gsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmin, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("gsum: %d, gmax: %d gmin:%d\n", gsum, gmax, gmin);
    MPI_Finalize();
}
```
Example: MPI_Allreduce

[Output]

```
[negrut@euler24 CodeBits]$ mpiexec -np 10 me759.exe
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |
| gsum: 45, gmax: 9, gmin: 0 |

[negrut@euler24 CodeBits]$
```
MPI_SCAN

- Performs a prefix reduction on data distributed across a communicator.

- The operation returns, in the receive buffer of the process with rank \( i \), the reduction of the values in the send buffers of processes with ranks \( 0, \ldots, i \) (inclusive).

- The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.
MPI_SCAN

before MPI_SCAN

- inbuf
- result

after

A  AoD  AoDoG  AoDoGoJ  AoDoGoJoM

done in parallel
Scan Operation

\[
\begin{array}{ccc}
A0 & B0 & C0 \\
A1 & B1 & C1 \\
A2 & B2 & C2 \\
\end{array}
\quad \xrightarrow{\text{scan}} \quad
\begin{array}{ccc}
A0 & B0 & C0 \\
A0+A1 & B0+B1 & C0+C1 \\
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
\end{array}
\]

data (input buffer) → data (output buffer)
**MPI_Scan: Prefix reduction**

- Process i receives data reduced on process 0 through i

```
MPI_Scan(sbuf, rbuf, 6, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
```

---

[A. Snavely]→
MPI_Scan

```c
int MPI_Scan (void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `comm` (communicator)

**Note:** `count` refers to total number of elements that will be received into receive buffer after operation is complete.
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n) {
        if (myRank == n) {
            for (i = 0; i < dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Scan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n) {
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    free(result); free(data_l);
    return 0;
}
**Example: MPI_Scan**

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.exe
Process 0. Entry: 0. Value: 0
Process 0. Entry: 1. Value: 0
Process 1. Entry: 0. Value: 1
Process 1. Entry: 1. Value: 2
Process 2. Entry: 0. Value: 2
Process 2. Entry: 1. Value: 4
Process 3. Entry: 0. Value: 3
```

```
Post Scan - Content on Process: 0
Entry: 0. Scan Val: 0
Entry: 1. Scan Val: 0
Post Scan - Content on Process: 1
Entry: 0. Scan Val: 1
Entry: 1. Scan Val: 2
Post Scan - Content on Process: 2
Entry: 0. Scan Val: 2
Entry: 1. Scan Val: 4
Post Scan - Content on Process: 3
Entry: 0. Scan Val: 3
Entry: 1. Scan Val: 6
Post Scan - Content on Process: 3
Entry: 0. Scan Val: 6
Entry: 1. Scan Val: 12
[negrut@euler26 CodeBits]$`
```
MPI_Exscan

- MPI_Exscan is like MPI_Scan, except that the contribution from the calling process is not included in the result at the calling process (it is contributed to the subsequent processes).

- The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not significant on process 0.

- The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0.

- For processes with rank \( i > 1 \), the operation returns, in the receive buffer of the process with rank \( i \), the reduction of the values in the send buffers of processes with ranks \( 0, \ldots, i-1 \) (inclusive).

- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.
int MPI_Exscan (void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Exscan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```
### Example: MPI_Exscan

[Output]

<table>
<thead>
<tr>
<th>Process 0. Entry: 0. Value: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 0. Entry: 1. Value: 0</td>
</tr>
<tr>
<td>Process 1. Entry: 0. Value: 1</td>
</tr>
<tr>
<td>Process 1. Entry: 1. Value: 2</td>
</tr>
<tr>
<td>Process 2. Entry: 0. Value: 2</td>
</tr>
<tr>
<td>Process 2. Entry: 1. Value: 4</td>
</tr>
<tr>
<td>Process 3. Entry: 0. Value: 3</td>
</tr>
</tbody>
</table>

Post Scan - Content on Process: 0
- Entry: 0. Scan Val: 321045752
- Entry: 1. Scan Val: 32593

Post Scan - Content on Process: 1
- Entry: 0. Scan Val: 0
- Entry: 1. Scan Val: 0

Post Scan - Content on Process: 2
- Entry: 0. Scan Val: 1
- Entry: 1. Scan Val: 2

Post Scan - Content on Process: 3
- Entry: 0. Scan Val: 3
- Entry: 1. Scan Val: 6
User-Defined Reduction Operations

- Operator handles
  - Predefined — see table of last lecture: MPI_SUM, MPI_MAX, etc.
  - User-defined

- User-defined operation ■:
  - Should be associative
  - User-defined function must perform the operation “vector_A ■ vector_B”

- Registering a user-defined reduction function:

  ```c
  MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
  ```

- `commute` tells the MPI library whether `func` is commutative or not
Example:
Norm 1 of a Vector

```c
#include <mpi.h>
#include <stdio>
#include <math.h>

void oneNorm(float *in, float *inout, int *len,
             MPI_Datatype *type)
{
    int i;
    for (i=0; i<*len; i++) {
        *inout = fabs(*in) + fabs(*inout); /* one-norm */
        in++;
        inout++;
    }
}
```

```c
int main(int argc, char* argv[]) {
    int root=0, p, myid;
    float sendbuf, recvbuf;
    MPI_Op myop;
    int commutes=1;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    //create the operator...
    MPI_Op_create(onenorm, commune, &myop);

    //get some fake data used to make the point...
    sendbuf = myid*(-1)^myid;
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Reduce (&sendbuf, &recvbuf, 1, MPI_FLOAT, myop, root, MPI_COMM_WORLD);
    if( myid == root )
        printf("The operation yields %f\n", recvbuf);
    MPI_Finalize();
    return 0;
}
```
```cpp
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

template <typename T>
struct absval {
    __host__ __device__
    T operator()(const T& x) const {
        return fabs(x);
    }
};

int main(void)
{
    // initialize host array
    float x[4] = {1.0, -2.0, 3.0, -4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    absval<float> unary_op;
    float res = thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, 0.f, thrust::plus<float>());

    std::cout << res << std::endl;
    return 0;
}
```
MPI Derived Types
[Describing Non-contiguous and Heterogeneous Data]
The Relevant Question

- The relevant question that we want to be able to answer?
  - “What’s in your buffer?”

- Communication mechanisms discussed so far allow send/recv of a contiguous buffer of identical elements of predefined data types

- Often want to send non-homogenous elements (structure) or chunks that are not contiguos in memory

- MPI enables you to define derived data types to answer the question “What’s in your buffer?”
MPI Datatypes

- MPI Primitive Datatypes
  - MPI_Int, MPI_Float, MPI_INTEGER, etc.

- Derived Data types - can be constructed by four methods:
  - contiguous
  - vector
  - indexed
  - struct
  - Can be subsequently used in all point-to-point and collective communication

- The motivation: create your own types to suit your needs
  - More convenient
  - More efficient
Type Maps

[Jargon]

- A derived data type specifies two things:
  - A sequence of primitive data types
  - A sequence of integers that represent the byte displacements, measured from the beginning of the buffer

- Displacements are not required to be positive, distinct, or in increasing order (however, negative displacements will precede the buffer)

- Order of items need not coincide with their order in memory, and an item may appear more than once
## Type Map

<table>
<thead>
<tr>
<th>Primitive datatype 0</th>
<th>Displacement of 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive datatype 1</td>
<td>Displacement of 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Primitive datatype n-1</td>
<td>Displacement of n-1</td>
</tr>
</tbody>
</table>
Extent

[Jargon]

- Extent: distance, in bytes, from beginning to end of type

- More specifically, the extent of a data type is defined as:
  … the span from the first byte to the last byte occupied by entries in this data type rounded up to satisfy alignment requirements

- Example:
  - Type=\{(\texttt{double},0), (\texttt{char},8)\} i.e. offsets of 0 and 8 respectively.
  - Now assume that doubles are aligned strictly at addresses that are multiples of 8
  - extent = 16 (9 rounds to next multiple of 8, which is where the next double would land)
Map Type, Examples

- What is extent of type \{\text{char, 0}, \text{double, 8}\}?
  Ans: 16

- Is this a valid type: \{\text{double, 8}, \text{char, 0}\}?
  Ans: yes, since order does not matter
Example

- What is Type Map of `MPI_INT`, `MPI_DOUBLE`, etc.?
  - `{(int,0)}`
  - `{(double, 0)}`
  - Etc.
The sequence of primitive data types (i.e. displacements ignored) is the type signature of the data type.

Example: a type map of

{\(\text{double}, 0\), \(\text{int}, 8\), \(\text{char}, 12\)}

...has a type signature of

{\text{double}, \text{int}, \text{char}}
Data Type Interrogators

- **datatype** - primitive or derived **datatype**
- **extent** - returns extent of **datatype** in bytes

```c
int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent);
```

- **datatype** - primitive or derived **datatype**
- **size** - returns size in bytes of the entries in the **type signature** of **datatype**
  - Gaps don’t contribute to size
  - This is the total size of the data in a message that would be created with this **datatype**
  - Entries that occur multiple times in the **datatype** are counted with their multiplicity

```c
int MPI_Type_size (MPI_Datatype datatype, int *size);
```
Committing Data Types

- Each derived data type constructor returns an uncommitted data type. Think of commit process as a compilation of data type description into efficient internal form

  ```c
  int MPI_Type_commit (MPI_Datatype *datatype);
  ```

- Required for any derived data type before it can be used in communication

- Subsequently can use in any function call where an MPI_Datatype is specified
MPI_Type_free

Call to MPI_Type_free sets the value of an MPI data type to MPI_DATATYPE_NULL

Data types that were derived from the defined data type are unaffected.

```c
int MPI_Type_free(MPI_Datatype *datatype);
```
MPI Type-Definition Functions
[“constructors”]

- **MPI_Type_Contiguous**: a replication of data type into contiguous locations
- **MPI_Type_vector**: replication of data type into locations that consist of equally spaced blocks
- **MPI_Type_create_hvector**: like vector, but successive blocks are not multiple of base type extent
- **MPI_Type_indexed**: non-contiguous data layout where displacements between successive blocks need not be equal
- **MPI_Type_create_struct**: most general – each block may consist of replications of different data types

- The inconsistent naming convention is unfortunate but carries no deeper meaning. It is a compatibility issue between old and new version of MPI.
MPI_Type_contiguous

```c
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- **IN** count  (replication count)
- **IN** oldtype  (base data type)
- **OUT** newtype (handle to new data type)

- Creates a new type which is simply a replication of old type into contiguous locations
```c
#include <stdio.h>
#include <mpi.h>

/* !!! Should be run with at least four processes !!! */

int main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;
        int y;
        int z;
    } point;
    MPI_Datatype ptype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if(rank==3) {
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    } else if(rank==1) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d) \n",rank,point.x,point.y,point.z);
    }
    MPI_Type_free(&ptype);
    MPI_Finalize();
    return 0;
}
```
Example: MPI_Type_contiguous

[Output]

{negrut@euler24 CodeBits} $ mpiexec -np 10 me759.exe
P:1 received coords are (15,23,6)
{negrut@euler24 CodeBits} $
Motivation: MPI_Type_vector

- Assume you have a 2D array of integers, and want send the last column

```c
int x[4][8];
```

Content of `x`:

```
  10  11  12  13  14  15  16
  100 101 102 103 104 105 106
1000 1001 1002 1003 1004 1005 1006
10000 10001 10002 10003 10004 10005 10006
```

- There should be a way to say that I want to transfer integers, 4 of them, and they are stored in array `x` 8 integers apart (the stride)
MPI_Type_vector: Example

- count = 2
- blocklength = 3
- stride = 5

oldtype

5 element stride between blocks

newtype

3 elements per block

2 blocks
MPI_Type_vector

- **MPI_Type_vector** is a constructor that allows replication of a data type into locations that consist of equally spaced blocks.

- Each block is obtained by concatenating the same number of copies of the old data type.

- Spacing between blocks is a multiple of the extent of the old data type.

- One way to look at it:
  - You want some entries but don’t care about other entries in an array.
  - There is a repeatability to this pattern of “wanted” and “not wanted” entries.
MPI_Type_vector

MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);

- IN  count  (number of blocks)
- IN  blocklength  (number of elements per block)
- IN  stride  (spacing between start of each block, measured as # elements)
- IN  oldtype  (base datatype)
- OUT newtype  (handle to new type)

- Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of oldtype with a stride that is multiple of extent of old type
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);

    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j)
                x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }

    MPI_Type_free(&coltype);
    MPI_Finalize();
    return 0;
}
Example: MPI_Type_vector

[Output]

Content of x:

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>1006</td>
<td>1007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10001</td>
<td>10002</td>
<td>10003</td>
<td>10004</td>
<td>10005</td>
<td>10006</td>
<td>10007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>100001</td>
<td>100002</td>
<td>100003</td>
<td>100004</td>
<td>100005</td>
<td>100006</td>
<td>100007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[negrut@euler19 CodeBits]$ mpiexec -np 12 me759.exe
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
[negrut@euler19 CodeBits]$
Example: MPI_Type_vector

- Given: Local 2D array of interior size $m \times n$ with $n_g$ ghostcells at each edge
- You wish to send the interior (non ghostcell) portion of the array
- How would you describe the data type to do this in a single MPI call?

```c
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

**Ans:**
```
MPI_Type_vector(m, n, n+2*ng, MPI_DOUBLE, &interior);
MPI_Type_commit(&interior);
MPI_Send(startPoint, 1, interior, dest, tag, MPI_COMM_WORLD);
```
Type Map Example

Start with \texttt{oldtype} for which
Type Map = \{(double, 0), (char, 8)\}

What is Type Map of \texttt{newtype} if defined as below?

\texttt{MPI\_Type\_vector}(2,3,4,\texttt{oldtype},&\texttt{newtype})

\texttt{MPI\_Type\_vector}(\texttt{int count, int blocklength, int stride, MPI\_Datatype oldtype, MPI\_Datatype *newtype});

Ans:

\{(double, 0), (char, 8)\}, \{(double,16),(char,24)\}, \{(double,32),(char,40)\},
\{(double,64),(char,72)\}, \{(double,80),(char,88)\}, \{(double,96),(char,104)\}
Exercise: MPI_Type_vector

- Express

  ```
  MPI_Type_contiguous(count, old, &new);
  ...
  ```

- As a call to `MPI_Type_vector`

  ```
  MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
  ```

- Ans:

  ```
  MPI_Type_vector (count, 1, 1, old, &new);
  MPI_Type_vector (1, count, count, old, &new);
  ```
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
In some MPI implementations there are more than 300 MPI functions
- Not all of them part of the MPI standard though, some vendor specific

Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code…
- MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize
The PETSc Library
[The message: Use libraries if available]

- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)
  - Open source, available for download at http://www.mcs.anl.gov/petsc/petsc-as/

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)
Structure of PETSc

PETSc PDE Numerical Solution Utilities

- ODE Integrators
- Visualization
- Nonlinear Solvers, Unconstrained Minimization
- Interface
- Linear Solvers
- Preconditioners + Krylov Methods
- Grid Management
- Object-Oriented Matrices, Vectors, Indices

Profiling Interface

Computation and Communication Kernels
- MPI, MPI-IO, BLAS, LAPACK
# PETSc Numerical Components

## Nonlinear Solvers
- **Newton-based Methods**
- **Line Search**
- **Trust Region**
- **Other**

## Time Steppers
- **Euler**
- **Backward Euler**
- **Pseudo Time Stepping**
- **Other**

## Krylov Subspace Methods
- **GMRES**
- **CG**
- **CGS**
- **Bi-CG-STAB**
- **TFQMR**
- **Richardson**
- **Chebychev**
- **Other**

## Preconditioners
- **Additive Schwartz**
- **Block Jacobi**
- **Jacobi**
- **ILU**
- **ICC**
- **LU** (Sequential only)
- **Others**

## Matrices
- **Compressed Sparse Row (AIJ)**
- **Blocked Compressed Sparse Row (BAIJ)**
- **Block Diagonal (BDIAG)**
- **Dense**
- **Matrix-free**
- **Other**

## Distributed Arrays

## Index Sets
- **Indices**
- **Block Indices**
- **Stride**
- **Other**

## Vectors
Flow Control for PDE Solution

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PC

KSP

PETSc

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

User code

PETSc code
CUDA, OpenMP, MPI: Putting Things in Perspective
Pros, CUDA

- Many remarkable success stories when the application targeted is data parallel and with high arithmetic intensity
  - One order of magnitude speed-ups are common

- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L got you six or seven years ago

- Ubiquitous
  - Present on more than 100 million computers today support CUDA

- Good productivity tools
Cons, CUDA

- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware.

- Not that much device memory available – 6 GB is the most you get today.
  - Getting around it requires moving data in and out of the device, which complicates the programming job.

- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance and complicating the implementation task.

- For true HPC, using CUDA in conjunction with MPI remains a challenge.
  - Ongoing projects aimed at addressing this, but still…
What Would Be Nice...

- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases.
- Integrate CPU & GPU so that concept of global device memory disappears.
- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming.
Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement.

- Programming model is “serial-like”, thus conceptually simpler than message passing.

- Compiler directives are generally simple and easy to use.

- Legacy serial code does not need to be rewritten.
Cons of OpenMP

- The model doesn’t scale up all that well

- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl’s Law prohibits substantial speedups

- Amdahl’s Law:
  \( s = \text{Fraction of serial execution time that cannot be parallelized} \)
  \( N = \text{Number of processors} \)

\[
\text{Execution speedup: } = \frac{1}{s + \frac{1-s}{N}}
\]

- If you have big loops that dominate execution time, these are ideal targets for OpenMP
Pros of MPI

- Good vendor support for the standard
  - It was great that the community converged upon a standard (something that can’t be said about GPU computing)

- Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores

- Can be deployed both for distributed as well as shared memory architectures

- Today it is synonym with High Performance Computing
  - Provided a clear and relatively straightforward framework for reaching Petaflops grade computing
Cons of MPI

- The interconnect is Achilles' heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically worse though

- Like CUDA, works well only for applications where you don’t have to communicate all that much (high arithmetic intensity)
General Remarks on Parallel Computing

- Parallel Computing is and will be relevant at least for this decade

- Nonetheless, it continues to be challenging

  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it’s a skill that is eventually acquired
    - Parallel Programming more difficult than programming for Sequential Computing

  - Productivity tools (debuggers, profilers, build solutions) more challenging to master

  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run
Skills I hope You Picked Up

- I think of these as items that you can add to your resume:
  - Basic understanding of hardware for parallel computing
  - Basic understanding of parallel execution models: SIMD, MIMD, etc.
  - CUDA programming
  - OpenMP Programming
  - MPI Programming
  - [ Build management: Cmake ]
  - Debugging: gdb, cuda-gdb, memcheck, cuda-memcheck
  - Profiling: nvvp
Most Important Two Things

- Don’t move data around
  - Costly in terms of time and energy

- Hone your “computational thinking” skills
  - Understand the hardware and how it works