Parallelism III
MPI, Vectorization, OpenACC, OpenCL

John Cavazos, Tristan Vanderbruggen, and Will Killian

Dept of Computer & Information Sciences
University of Delaware
Lecture Overview

- Introduction
- MPI
- Vectorization
- OpenACC
- OpenCL
- Conclusion / Q&A
3 - MPI

- Model
- Language
- Step-by-step Example
- Q&A
3.1 - MPI: Model

- Distributed Memory, originally
- today implementation support shared memory SMP

Source: https://computing.llnl.gov/tutorials/mpi/
3.2 - MPI: Language

- **MPI is an Interface**
  - MPI = Message Passing Interface
- **Different implementations are available for C / Fortran**

<table>
<thead>
<tr>
<th><strong>C Binding</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format:</strong></td>
<td>rc = MPI_Xxxxxx(parameter, ...)</td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td>rc = MPI_Bsend(&amp;buf, count, type, dest, tag, comm)</td>
</tr>
<tr>
<td><strong>Error code:</strong></td>
<td>Returned as &quot;rc&quot;. MPI_SUCCESS if successful</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Fortran Binding</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format:</strong></td>
<td>CALL MPI_XXXXXX(parameter,..., ierr)</td>
</tr>
<tr>
<td></td>
<td>call mpi_xxxxxx(parameter,..., ierr)</td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td>CALL MPI_BSEND(buf, count, type, dest, tag, comm, ierr)</td>
</tr>
<tr>
<td><strong>Error code:</strong></td>
<td>Returned as &quot;ierr&quot; parameter. MPI_SUCCESS if successful</td>
</tr>
</tbody>
</table>

Source: https://computing.llnl.gov/tutorials/mpi/
3.3 - MPI: Step-by-step Example

General MPI Program Structure:

1. **MPI include file**
   - Declarations, prototypes, etc.
   - Program begins
   - **Serial code**

2. **Initialize MPI environment**
   - Parallel code begins

3. **Do work & make message passing calls**

4. **Terminate MPI environment**
   - Parallel code ends
   - **Serial code**

5. Program ends

Source: https://computing.llnl.gov/tutorials/mpi/
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Get_processor_name(hostname, &len);

    printf ("Hello from task %d on %s!
", taskid, hostname);

    if (taskid == 0)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

    MPI_Finalize();

    return 0;
}
```
3.3 (a) - MPI: Hello World

- Compile
  - `mpicc helloworld-mpi.c -o helloworld-mpi`
  - OR
  - `gcc -c helloworld-mpi.c -o helloworld-mpi.o`
  - `mpicc helloworld-mpi.o -o helloworld-mpi`
  - **Warning:** Select the good toolchain!
3.3 (a) - MPI: Hello World

• Run
  o On one node:
    ▪ `mpirun -n $NB_PROCCESS ./helloworld-mpi`
  o On a cluster with `qsub` (Sun Grid Engine)
    ▪ `qsub -pe mpich $NB_PROCCESS mpi-qsub.sh`
    ▪ With `mpi-qsub.sh`:

```bash
#!/bin/bash
#
#$ -cwd
#
mpirun -np $NSLOTS ./matmul-mpi
```
3.3 (b) - MPI: Matrix Multiply

```c
#define NRA N    /* number of rows in matrix A */
#define NCA P    /* number of columns in matrix A */
#define NCB M    /* number of columns in matrix B */
#define MASTER 0 /* taskid of first task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */

int numtasks,                  /* number of tasks in partition */
taskid,                       /* a task identifier */
numworkers,                   /* number of worker tasks */
source,                       /* task id of message source */
dest,                        /* task id of message destination */
mtype,                       /* message type */
rows,                        /* rows of matrix A sent to each worker */
averow, extra, offset,       /* used to determine rows sent to each worker */
i, j, k, rc;                 /* misc */

double a[NRA][NCA],          /* matrix A to be multiplied */
b[NCA][NCB],                 /* matrix B to be multiplied */
c[NRA][NCB];                 /* result matrix C */

MPI_Status status;
```
MPI initialization:

```c
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
if (numtasks < 2 ) {
    printf("Need at least two MPI tasks. Quitting...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
}
numworkers = numtasks-1;
```
3.3 (b) - MPI: Matrix Multiply

Master initialization:

```c
if (taskid == MASTER)
{
    printf("mpi_mm has started with %d tasks.\n", numtasks);
    printf("Initializing arrays...\n");
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j] = i+j;
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j] = i*j;
```
3.3 (b) - MPI: Matrix Multiply

```c
/* Send matrix data to the worker tasks */
averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0;
mttype = FROM_MASTER;
for (dest=1; dest<=numworkers; dest++)
{
    rows = (dest <= extra) ? averow+1 : averow;
    printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
}
```
/* Receive results from worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++)
{
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
              MPI_COMM_WORLD, &status);
    printf("Received results from task %d\n",source);
}
3.3 (b) - MPI: Matrix Multiply

```c
if (taskid > MASTER)
{
    mtype = FROM_MASTER;
   (MPI_RECV(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_RECV(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_RECV(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_RECV(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++)
            c[i][k] = 0.0;
    for (j=0; j<NCA; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];

    mtype = FROM_WORKER;
    MPI_SEND(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_SEND(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_SEND(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
}
MPI_Finalize();
```
3.3 (b) - MPI: Matrix Multiply

1 - On master after initialization

2 - On worker after comm

3 - On worker after computation

4 - On master after comm
Vectorization

- Newer CPUs have advanced Instruction Set Architectures
- Vector Processing Units
  - Introduced with Intel Pentium III
  - Subsequently expanded every year
- Single operation working on multiple data values (SIMD)
- Method of parallelization done at the hardware level
Overview of SISD vs SIMD
Intel SSE

- SSE (Streaming SIMD Extensions)
- Supported on every new Intel and AMD Processor
- 128-bit registers
  - Allows for 4 floating-point numbers to be operated on with one instruction
  - Alternatively can modify 2 double-precision floating-point numbers

<table>
<thead>
<tr>
<th>f3</th>
<th>f2</th>
<th>f1</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>96 - 127</td>
<td>64 - 95</td>
<td>32 - 63</td>
<td>0 - 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d1</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 - 127</td>
<td>0 - 63</td>
</tr>
</tbody>
</table>
• Basic Instructions
  • Addition, Multiplication, Subtraction, Division, Reciprocal

instrps %xmm1, %xmm0
Instruction 128-bit register 128-bit register

addps subps mulps divps
Example of more complicated instruction (shuffle)

```c
shufps %xmm0, %xmm1, mask
```

**Mask Properties**

- Index of 1st register to store
- Index of 1st register to store
- Index of 2nd register to store
- Index of 2nd register to store
Intrinsics

- Wrapper functions to direct assembly calls
- Programmer has control
- Benefits when used with SIMD Instructions
- C-like function calls

```c
float dot (const __m128 &x1, const __m128 &x2)
{
    __m128 res;
    res = _mm_mul_ps (x1, x2);    // multiply same index
    res = _mm_hadd_ps (res, res); // add u- and l-words
    res = _mm_hadd_ps (res, res); // add u- and l-words
    return _mm_cvtss_f32 (res);   // obtain float value
}
```
There have been many different Vector processing releases on microprocessors.

SSE, SSE2, SSE3, SSSE3, SSE4, SSE4a, SSE4.1, SSE4.2, AVX, AVX2

How can we have the fastest execution time?

- Implement specialization for each target vector processing level
- At compilation time, the fastest (highest level) implementation is compiled
float
dot (const __m128 &x1, const __m128 &x2)
{
    __m128 res;
    #ifdef __SSE4_1__
        res = _mm_dp_ps (x1, x2, 0xFF);
    #else
    #ifdef __SSE3__
        res = _mm_mul_ps (x1, x2);
        res = _mm_hadd_ps (res, res);
        res = _mm_hadd_ps (res, res);
    #else
        // fallback – SSE2
        __m128 m = _mm_mul_ps (x1, x2);
        __m128 t = _mm_add_ps (m, _mm_shuffle_ps(m, m, 0xB1));
        res = _mm_add_ps(t, _mm_shuffle_ps(t, t, 0x4E));
    #endif
    #endif
    return _mm_cvtss_f32 (res);
}
Vectorization

- Applications
  - Physics Simulations
  - Linear Algebra Computations

- Background required?
  - Extremely high!
  - Need to know architecture of systems
  - Need to account for arbitrary problem sizes

- Solution?
  - Libraries already exists!
Eigen

- C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms

- Eigen applies vectorization to your operations automatically
• Versatile
  • Supports all matrix sizes (small, fixed size up to arbitrarily large dense or sparse)
  • Supports all types (float, double, int)
  • Supports various matrix decompositions and geometry features
  • Has unsupported modules for non-linear optimization, FFT, polynomial solver
• Fast
  • Expression templates in C++ remove temporary variables
  • Explicit vectorization is automatically performed on SSE and ARM NEON
  • Fixed-size matrices are fully optimized (dynamic memory avoided, loops unrolled)
  • Large matrices observe “cache friendliness”
• Reliable
  • Thoroughly tested
  • Algorithms are selected for reliability
    • All tradeoffs are explicitly listed
• Elegant
  • Easy-to-use API, natural for C++ programmers
  • Implementing an algorithm is “like copying pseudocode”
Eigen

- Open Source
- Freely available for download

- Limitations:
  - C++ only

Website: [http://eigen.tuxfamily.org](http://eigen.tuxfamily.org)
```cpp
#include <cstdio>
#include <cstdlib>
#include <time.h>
#include <sys/time.h>

#include <Eigen/Dense>

const unsigned int SIZE = 256;
const unsigned int ROWS1 = SIZE;
const unsigned int COLS1 = SIZE;

const unsigned int ROWS2 = SIZE;
const unsigned int COLS2 = SIZE;
```
int main () {
    float M1 [ROWS1][COLS1];
    float M2 [ROWS2][COLS2];
    Eigen::MatrixXf m1 (ROWS1, COLS1);
    Eigen::MatrixXf m2 (ROWS2, COLS2);
    for (unsigned int i = 0; i < ROWS1; ++i)
        for (unsigned int j = 0; j < COLS1; ++j)
            { m1 (i, j) = (i + j) % 32;
              M1 [i][j] = (i + j) % 32;
            }
    for (unsigned int i = 0; i < ROWS2; ++i)
        for (unsigned int j = 0; j < COLS2; ++j)
            { m2 (i, j) = (i - j) % 32;
              M2 [i][j] = (i - j) % 32;
            }
}
Vectorization Example

Eigen::MatrixXf m3 = m1 * m2;

float M3[ROWS1][COLS2];
for (unsigned int r = 0; r < ROWS1; ++r) {
    for (unsigned int c = 0; c < COLS2; ++c) {
        M3[r][c] = 0;
        for (unsigned int i = 0; i < COLS1; ++i) {
            M3[r][c] += M1[r][i] * M2[i][c];
        }
    }
}

g++ -O2 -march=native -DEIGEN_NO_DEBUG eigen_mm.cpp -I/include/
Vectorization

- Questions / Comments?
OpenACC

- Directive-based programming model that targets accelerators (GPUs) instead of CPUs
- Set of standardized, high-level pragmas that enable C/C++ and Fortran programmers to code on GPU with familiarity of OpenMP
- Supporting companies are currently NVIDIA, PGI, CAPS Enterprise, and Cray
OpenACC Syntax

- Syntax extremely similar to OpenMP

**C/C++**

```c
#pragma acc directive-name [clause [[, clause]...] new-line
```

**Fortran**

```fortran
!$acc directive-name [clause [[, clause]...] new-line
```
OpenACC Pragmas

- Directives:
  - kernels, data, loop, wait

- Clauses:
  - if, async, private, firstprivate, reduction, deviceptr
  - copy, copyin, copyout, create, present, present_or_copy, present_or_copyin, present_or_copyout
  - gang, worker, vector, num_gangs, num_workers, vector_length, seq, collapse
Device Information

acc_get_num_devices()
acc_set_device_type()
acc_get_device_type()
acc_set_device_num()
acc_get_device_num()
Synchronization

acc_async_test()
acc_async_test_all()
acc_async_wait()
acc_async_wait_all()
Runtime Routines

OpenACC Runtime:
acc_init()
acc_shutdown()

Memory Routines:
acc_on_device()
acc_malloc()
acc_free()
#define SIZE = 1000;
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

int main()
{
    int i, j, k;
    // initialize
    for (i = 0; i < SIZE; ++i)
    {
        for (j = 0; j < SIZE; ++j)
        {
            // populate with some values
            a[i][j] = (float)i + j;
            b[i][j] = (float)i - j;
            c[i][j] = 0.0f;
        }
    }
}
```c
// Compute matrix multiplication.
#pragma acc kernels copyin(a,b) copy(c)
for (i = 0; i < SIZE; ++i)
{
    for (j = 0; j < SIZE; ++j)
    {
        for (k = 0; k < SIZE; ++k)
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
return 0;
```

OpenACC Directives:
- Tells OpenACC to generate a device kernel.
- Copies a and b to the device.
- Copies c to the device then transfers back over to host once execution finishes.
- Tells OpenACC to generate a device kernel.
Compiling OpenACC

- HMPP Workbench 3.2.1
  - Available from CAPS Enterprise
  - Trial Version available
- Alternative: PGI compiler
  - 14 day OpenACC trial available
- Currently no free compiler available

```
hmpp <hmpp_options> <cc> <cflags> <source>
```

Ex). `hmpp -codelet-required gcc -o mm_acc mm_acc.c`
Improving Data Transfer

• Initialized data on host, then transferred to device
• Just allocate data on device to reduce to one transfer

```c
int i, j, k;
#pragma acc kernels create(a,b) copyout(c)
{
    for (i = 0; i < SIZE; ++i) {
        for (j = 0; j < SIZE; ++j) {
            // populate with some values
            a[i][j] = (float)i + j;
            b[i][j] = (float)i - j;
            c[i][j] = 0.0f;
        }
    }
    // compute code goes here ...
}
```
• First Example:
  • Generated 1 kernel for matrix multiplication
    • Three memory transfers into kernel
    • One memory transfer out

• Second Example:
  • Generated 2 kernels for matrix multiplication
    • Kernel #1:
      • Allocated memory on device and initialized values
    • Kernel #2:
      • Performed multiplication and transferred result to host
Accelerator Memory Model

- Location can explicitly be specified
  - global or local
- Larger memory size causes reduced throughput
  - Keep data as local as possible
- Memory size restrictions
  - Usually < 4GB total on accelerators
Memory Model

Accessible by all work-items
Memory Model

Compute Device

Compute unit 1

Private memory 1

Private memory M

PE 1

PE M

Local memory 1

Compute unit N

Private memory 1

Private memory M

PE 1

PE M

Local memory N

Global/Constant Memory Data Cache

Global Memory

Constant Memory

Compute Device Memory

Read-Only Global Memory
Memory Model

Local to a work-group
Memory Model

Private to a work-item
• Executes on one or more Processing Elements

• Three Levels:
  • Gang
  • Worker
  • Thread
• In a given parallel region the following occurs:
  • Gangs of workers threads are created to execute on accelerator
  • One worker in each gang begins executing the code following the structured block
• Within a parallel region
  • specifies that the loop iteration need to be distributed among gangs.

• Within a kernel region
  • loop iteration need to be distributed among gangs
  • also used to specify how many gangs will execute the iteration of a loop
Worker

- Within a parallel region
  - specifies that the loop iteration need to be distributed among workers of a gang.
- Within a kernel region
  - loop iteration need to be distributed among workers of a gang
  - also used to specify how many workers of a gang will execute the iteration of a loop
Vector

- Within a parallel region:
  - specifies that the loop iterations need to be in vector or SIMD mode
  - uses the vector length specified by the parallel region

- Within a kernel region:
  - specifies that the loop iterations need to be in vector or SIMD mode.
  - if an argument is specified, the iterations will be processed in vector strips of that length.
• Questions / Comments?
Conclusion

- Final Questions?