Lecture 13
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Message Passing
Stencil methods with message passing
Announcements

• Weds office hours changed for the remainder of quarter: 3:30 to 5:30
Today’s lecture

• Aliev Panfilov Method (A3)
• Message passing
• Stencil methods in MPI
Warp aware summation

For next time:
complete the code so it handles global data with arbitrary N

\[
\text{reduceSum} \llll <N/512,512> \ggg (x,N)
\]

```c
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + tid;
    unsigned int s;

    for (s = blockDim.x/2; s > 1; s /= 2) {
        __syncthreads();
        if (tid < s )
            x[tid] += x[tid + s ];
    }

    if (tid == 0) atomicAdd(total,x[tid]);
}
```
Recapping from last time

• Stencil methods use nearest neighbor computations
• The Aliev-Panfilov method solves a coupled set of differential equations on a mesh
• We showed how to implement it on a GPU
• We use shared memory (and registers) to store “ghost cells” to optimize performance
Computational loop of the cardiac simulator

- **ODE solver:**
  - No data dependency, trivially parallelizable
  - Requires a lot of registers to hold temporary variables

- **PDE solver:**
  - Jacobi update for the 5-point Laplacian operator.
  - Sweeps over a uniformly spaced mesh
  - Updates voltage to weighted contributions from the 4 nearest neighbors updating the solution as a function of the values in the previous time step

For a specified number of iterations, using supplied initial conditions repeat

```plaintext
for (j=1; j < m+1; j++){
  for (i=1; i < n+1; i++) {
    // PDE SOLVER
    E[j,i] = E_p[j,i]+α*(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
    // ODE SOLVER
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
    R[j,i] += dt*(ε+M1*R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
  }
} swap E_p and E
End repeat
```
Where is the time spent (Sorken)?

- Loops are unfused

I1 cache: 32768 B, 64 B, 8-way associative
D1 cache: 32768 B, 64 B, 8-way associative
LL cache: 20971520 B, 64 B, 20-way associative
Command: ./apf -n 256 -i 2000

Ir I1mr I1mr Ilmr Dr D1mr D1mr D1mr Dw D1mw DLmw
4.451B 2,639 2,043 1,381,173,237 50,592,465 7,051 3957M 16,794,937 26,115 PROGRAM TOTALS

Dr D1mr
1,380,464,019 50,566,007 solve.cpp:solve( ...)
. . . // Fills in the TOP Ghost Cells
10,000 1,999 for (i = 0; i < n+2; i++)
516,000 66,000 E_prev[i] = E_prev[i + (n+2)*2];
// Fills in the RIGHT Ghost Cells
10,000 0 for i = (n+1); i < (m+2)*(n+2); i+=(m+2)
516,000 504,003 E_prev[i] = E_prev[i-2];
// Solve for the excitation, a PDE
1,064,000 8,000 for(j =innerBlkRowStartIndx; j<=innerBlkRowEndIndx; j+=(m+)){
0 0 E_prevj = E_prev + j; E_tmp = E + j;
512,000 0 for(i = 0; i < n; i++) {
721,408,002 16,630,001 E_tmp[i] = E_prevj[i]+alpha*(E_prevj[i+1]...)
}
// Solve the ODEs
4,000 4,000 for(j=innerBlkRowStartIndx; j <= innerBlkRowEndIndx; j+=(m+3)){
262,144,000 33,028,000 for(i = 0; i <= n; i++) {
393,216,000 4,000 E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)...) *R_tmp[i];
393,216,000 4,000 R_tmp[i] += dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);
Fusing the loops

• On Sorken
  - Slows down the simulation by 20%
  - # data references drops by 35%
  - total number of L1 read misses drops by 48%

• What happened?

• Code didn’t vectorize

For a specified number of iterations, using supplied initial conditions repeat
for (j=1; j < m+1; j++){
  for (i=1; i < n+1; i++) {
    // PDE SOLVER
    E[j,i] = E_p[j,i]+α*(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
    // ODE SOLVER
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
    R[j,i] += dt*(ε+M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
  }
}
swap E_p and E
End repeat
Vectorization output

- Gcc compiles with `-ftree-vectorizer-verbose=1`

Analyzing loop at solve.cpp:118

solve.cpp:43: note: vectorized 0 loops in function.

For a specified number of iterations, using supplied initial conditions repeat

```c
for (j=1; j < m+1; j++) {
    for (i=1; i < n+1; i++) {
        // PDE SOLVER
        E[j,i] = E_p[j,i]+α*(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
        // ODE SOLVER
        E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
        R[j,i] += dt*(ε+M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
    }
}
swap E_p and E
End repeat
```
On Stampede

- We use the Intel compiler suite
  icpc --std=c++11  -O3   -qopt-report=1  -c  solve.cpp
  icpc: remark #10397: optimization reports are generated in
       *.optrpt files in the output location

LOOP BEGIN at solve.cpp(142,9)

  remark #25460: No loop optimizations reported

LOOP END

for (j=1; j < m+1; j++) {
    for (i=1; i < n+1; i++) {   // Line 142
        // PDE SOLVER
        E[j,i] = E_p[j,i]+α*(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);
        // ODE SOLVER
        E[j,i] += -dt*(kk*E[j,i]-(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
        R[j,i] +=  dt*(ε+M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
    }
}
A vectorized loop

• We use the Intel compiler suite
  icpc --std=c++11 -O3 -qopt-report=1 -c solve.cpp

6: for (j=0; j< 10000; j++)
   x[j] = j-1;
8: for (j=0; j< 10000; j++)
   x[j] = x[j]*x[j];

LOOP BEGIN at vec.cpp(6,3)
remark #25045: Fused Loops: (6 8)

remark #15301: FUSED LOOP WAS VECTORIZED
LOOP END
Thread assignment in a GPU implementation

• We assign threads to interior cells only
• 3 phases
  1. Fill the interior
  2. Fill the ghost cells – red circles correspond to active threads, orange to ghost cell data they copy into shared memory
  3. Compute – uses the same thread mapping as in step 1
Today’s lecture

- Aliev Panfilov Method (A3)
- Message passing
  - The Message Passing Programming Model
  - The Message Passing Interface - MPI
  - A first MPI Application – The Trapezoidal Rule
- Stencil methods in MPI
Architectures without shared memory

• *Shared nothing* architecture, or a *multicomputer*
• Hierarchical parallelism

Wikipedia

uk.hardware.info
tinyurl.com/k6jqag5

Scott B. Baden / CSE 260, UCSD / Fall '15
Programming with Message Passing

• Programs execute as a set of \( P \) processes (user specifies \( P \))
• Each process assumed to run on a different core
  - Usually initialized with the same code, but has private state
    \( \text{SPMD} = \) “Same Program Multiple Data”
  - Access to local memory only
  - Communicates with other processes by passing messages
  - Executes instructions at its own rate according to its \textit{rank} \((0:P-1)\) and the messages it sends and receives
Bulk Synchronous Execution Model

- A process is either communicating or computing
- Generally, all processors are performing the same activity at the same time
- Pathological cases, when workloads aren’t well balanced
Message passing

• There are two kinds of communication patterns

• *Point-to-point* communication:
  a single pair of communicating processes copy data between address space

• *Collective communication*: all the processors participate, possibly exchanging information
Point-to-Point communication

- Messages are like email; to send or receive one, we specify
  - A destination and a message body (can be empty)
- Requires a sender and an explicit recipient that must be aware of one another
- Message passing performs two events
  - Memory to memory block copy
  - Synchronization signal at recipient: “Data has arrived”
Send and Recv

- Primitives that implement Pt to Pt communication
- When **Send( )** returns, the message is “in transit”
  - Return doesn’t tell us if the message has been received
  - The data is somewhere in the system
  - Safe to overwrite the buffer
- **Receive( )** blocks until the message has been received
  - Safe to use the data in the buffer

<table>
<thead>
<tr>
<th>Send(y, 1)</th>
<th>Recv(x)</th>
</tr>
</thead>
</table>

Scott B. Baden / CSE 260, UCSD / Fall '15 24
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- If different processes send messages to the same destination, the order of receipt isn’t defined across sources.
Today’s lecture

• Aliev Panfilov Method (A3)
• Message passing
  • The Message Passing Programming Model
  • The Message Passing Interface - MPI
  • A first MPI Application – The Trapezoidal Rule
• Stencil methods in MPI
MPI

• We’ll use a library called MPI “Message Passing Interface”
  • 125 routines in MPI-1
  • 7 minimal routines needed by every MPI program
    • start, end, and query MPI execution state (4)
    • non-blocking point-to-point message passing (3)

• Reference material:
  
cseweb.ucsd.edu/~baden/Doc/mpi.html
Functionality we’ll cover today

• Point-to-point communication
• Message Filtering
• Communicators and Tags
• Application: the trapezoidal rule
• Collective Communication
A first MPI program: “hello world”

#include "mpi.h"

int main(int argc, char **argv ){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Hello, world! I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return(0);
}
Send andRecv

\texttt{const int Tag=99;}
\texttt{int msg[2] = \{ rank, rank \ast rank\};}
\texttt{if (rank == 0) \{ }
\begin{itemize}
\item \texttt{MPI_Status status;}
\item \texttt{MPI_Recv(msg, 2, MPI_INT, 1, Tag, MPI_COMM_WORLD, \&status);}
\end{itemize}
\texttt{\}}
\texttt{else MPI_Send(msg, 2, MPI_INT, 0, Tag, MPI_COMM_WORLD);}
Communicators

- A communicator is a name-space (or a context) describing a set of processes that may communicate
- MPI defines a default communicator `MPI_COMM_WORLD` containing all processes
- MPI provides the means of generating uniquely named subsets (later on)
- A mechanism for screening or filtering messages
MPI Tags

- Tags enable processes to organize or screen messages
- Each sent message is accompanied by a user-defined integer tag:
  - Receiving process can use this information to organize or filter messages
  - `MPI_ANY_TAG` inhibits tag filtering
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
  ◆ The length in bytes is \texttt{sizeof(type)} \times \# elements
  ◆ We don’t specify the as the \# byte
• MPI specifies a set of built-in types for each of the primitive types of the language
• In C: \texttt{MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,…}
• Also defined types, e.g. structs
Message status

• An MPI_Status variable is a struct that contains the sending processor and the message tag
• This information is useful when we aren’t filtering messages
• We may also access the length of the received message (may be shorter than the message buffer)

    MPI_Recv( message, count,
               TYPE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, COMMUNICATOR,
               &status);

    MPI_Get_count(&status, TYPE, &recv_count);

    status.MPI_SOURCE    status.MPI_TAG
Today’s lecture

• Aliev Panfilov Method (A3)

• Message passing
  ◆ The Message Passing Programming Model
  ◆ The Message Passing Interface - MPI
  ◆ A first MPI Application – The Trapezoidal Rule

• Stencil methods in MPI
The trapezoidal rule

- Use the trapezoidal rule to numerically approximate a definite integral, area under the curve.
- Divide the interval \([a, b]\) into \(n\) segments of size \(h = 1/n\).
- Area under the \(i^{th}\) trapezoid
  \[\frac{1}{2} (f(a+i\times h)+f(a+(i+1)\times h)) \times h\]
- Area under the entire curve
  \[\approx \sum \text{of all the trapezoids}\]
Reference material

• For a discussion of the trapezoidal rule
  http://en.wikipedia.org/wiki/Trapezoidal_rule

• A applet to carry out integration

• Code on Bang/Sorken/Stampede
  (from Pacheco’s MPI text)

  **Serial Code**
  $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/serial.c

  **Parallel Code**
  $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/trap.c
main() {
    float f(float x) { return x*x; } // Function we're integrating

    float h = (b-a)/n; // h = trapezoid base width
    // a and b: endpoints
    // n = # of trapezoids

    float integral = (f(a) + f(b))/2.0;

    float x; int i;

    for (i = 1, x=a; i <= n-1; i++) {
        x += h;
        integral = integral + f(x);
    }
    integral = integral*h;
}
Parallel Implementation of the Trapezoidal Rule

- Decompose the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- Processors combine their local integrals into a global one
First version of the parallel code

int local_n = n/p;       // # trapezoids; assume p divides n evenly
float local_a = a + my_rank*local_n*h,
local_b = local_a + local_n*h,
integral = Trap(local_a, local_b, local_n);

if (my_rank == ROOT) {   // Sum the integrals calculated by
                          // all processes
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, MPI_ANY_SOURCE,
                  tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, WORLD);