ASPhALT
An Automatic System for Parallel AppLication Transformation

Martin Swany
Application Transformation

- Parallel application performance depends on efficient data movement
- Programming methodologies that can yield good performance can be tedious and difficult
  - Asynchronous and one-sided
  - Specifics of network interfaces change
- Our work focuses on program transformations to reduce the cost of communication
Overlapping Computation and Communication

(a) Non-blocking, CPU-based I/O

- CPU
  - Application
  - OS Kernel
  - TCP/IP
  - NIC

(b) Asynchronous, RDMA-based I/O

- CPU
  - Application
  - RAM
  - NIC

Execution, Data Transfer, Overlapping, Inter-Device Operation
Overlapping Details

• Minimize overhead of data movement by overlapping it with useful work
  • An well-known idea
• What does it mean for parallel application structure?
  • Post a send as soon as the data is ready (without copying, if possible)
  • Do useful work
  • Check status after completion (with minimal polling or sleeping)
• Difficult to optimize, difficult to maintain
  • Not portable across platforms
Basic Approach

• Compiler-based application transformation
  – Source to source

• Transform MPI communication
  – Collectives \(\rightarrow\) Point-to-point
  – Blocking \(\rightarrow\) Non-blocking
  – Non-blocking \(\rightarrow\) One-sided

• Use analytical methods to reduce search space
  – Understanding network characteristics

• Use empirical techniques to refine
  – Too many factors affect overall application performance
System Structure
ASPhALT Framework

• Early work was based on Nestor and was Fortran-only (CompUniFormer)
• ASPhALT is based on the Open64 compiler
  • open64.sourceforge.net maintained by UD ECE
• Open64 has a fairly well-defined intermediate representation known as WHIRL
  – A WHIRL tree can be transformed and unparsed to high-level source code
Overlapping Transformation Example

Original code

```plaintext
integer, dimension(M,N):: array

do i = 1, N
   /* computation kernel */
   subroutine( array(1,i) )
endo

size = M*N
DataTransferCall( array(1,1), size, ... )
Other_Computation()
```

Tiled code

```plaintext
integer, dimension(M,N):: array

do i = 1, N, K
   do j = i, i+K-1
      /* computation kernel */
      subroutine( array(1,j) )
endo
   if( i > K ) then
      /* block for the arrival of the data */
      MPI_WAITALL( request( i - K ) )
   endif

size = M*K
/* asynchronous network transfer */
MPI_ISSEND( array(1,i), size, ... )
MPI_IRecv( dest(...), request(i), ... )
endo

MPI_WAITALL( request( i - K ) )
```
Evaluation

- Original Fortran code using MPI_ALLTOALL
- \texttt{ir\_transform} transforms code into version with aggressive early sending
  - Non-blocking using \texttt{isend/irecv}
  - Parameters chosen manually (only tile size here)
- Unparsed to Fortran
- To compare, we created communicationless versions of the code
  - Normalized execution time

\[
\begin{array}{c}
\text{ExperimentRuntime} \\
\text{CommunicationlessRuntime}
\end{array}
\]

From: Danalis, Pollock, Swany, \textit{in submission}
Automatic Transformation (SCI from Dolphin, NP=16)
Automatic Transformation
(iWARP Eth by Amasso, NP=24)
Comparison of the Code

<table>
<thead>
<tr>
<th>Scheme</th>
<th>LOC</th>
<th>Synchronization</th>
<th>Additional Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>19</td>
<td>Implicit</td>
<td>None</td>
</tr>
<tr>
<td>Tiled</td>
<td>73</td>
<td>Call to MPI_WAITALL()</td>
<td>None</td>
</tr>
<tr>
<td>Tiled &amp; one-sided I/O</td>
<td>190</td>
<td>Message padding, or additional messages</td>
<td>Buffer Registration &amp; Initial Handshake</td>
</tr>
</tbody>
</table>

Automatic transformation is a good thing!
Current Work

• Apply technique to Scatter/Gather code (C)
  • Early results show speedup for this case
• Apply technique to tiling large send (also C)
  • Again, early results are promising
  • Obviously, matching sends/recvs difficult without out of band information
• Addition of annotation
• Use OpenFabrics APIs
  • DAPL
Next Steps

- Empirical optimization framework
  - Simple approach: Generate various versions and run the code
- Investigate “compiled communication” MPI
  - Inline basic functionality when possible
Conclusion

- Profitable transformations for simple problems
- Very desirable to expand system to improve data movement performance
  - Optimizing this by hand is hard
  - The problem gets worse for large machines
- We must investigate the interaction between these transformations and local ones
Acknowledgements

• UD Students
  – Anthony Danalis, Andrew Gearheart
    • (alumni: Lewis Fishgold, Kiyong Kim)

• co-PI: Lori Pollock

• NSF CSR Program
  – Advanced Execution Systems area