Introduction

- Until recently, innovations in processor technology have resulted in computers with CPUs that operate at higher clock rates.
- However, as clock rates approach their theoretical physical limits, companies are developing new processors with multiple processing cores.
- With these multicore processors the best performance and highest throughput is achieved by using parallel programming techniques.

This requires knowledge and suitable programming tools to take advantage of the multicore processors.
Multiprocessors & Multicore Processors

Multiprocessor systems contain multiple CPUs that are not on the same chip.

The multiprocessor system has a divided cache with long-interconnects.

Multicore Processors contain any number of multiple CPUs on a single chip.

The multicore processors share the cache with short interconnects.
Hyper-threading

- Hyper-threading is a technology that was introduced by Intel, with the primary purpose of improving support for multi-threaded code.
- Under certain workloads hyper-threading technology provides a more efficient use of CPU resources by executing threads in parallel on a single processor.
- Hyper-threading works by duplicating certain sections of the processor.
- A hyper-threading equipped processor (core) pretends to be two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.
- E.g. Pentium 4, Xeon, Core i5 and Core i7 processors implement hyper-threading.
FPGAs

- FPGA = Field Programmable Gate Array
  - VHDL for programming of FPGAs is part of FYS4220!
- Contains huge amount of programmable gates that can be programmed into many parallel hardware paths
- FPGAs are truly parallel in nature so different processing operations do not have to compete for the same resources (no thread prioritization as typical to most common operating system)
How LabVIEW Implements Multithreading

- Parallel code paths on a block diagram can execute in unique threads
- LabVIEW automatically divides each application into multiple execution threads (originally introduced in 1998 with LabVIEW 5.0)
Two separate tasks that are not dependent on one another for data will run in parallel without the need for any extra programming.
How LabVIEW Implements Multithreading II

- Automatic Multithreading using LabVIEW Execution System (Implicit Parallelism / Threading)

1. LabVIEW compiler analyzes diagram and assigns code pieces to “clumps”

2. Information about which pieces of code can run together are stored in a run queue

3. If block diagram contains enough parallelism, it will simultaneously execute in all system threads

# of threads scales based on # of CPUs
Multicore Programming Goals

• Increase code execution speed (# of FLOPS)
• Maintain rate of execution but increase data throughput
• Evenly balance tasks across available CPUs (fair distribution of processing load)
• Dedicate time-critical tasks to a single CPU
Data Parallelism

You can speed up processor-intensive operations on large data sets by segmenting the data.
Example: Data Parallelism in LabVIEW

A standard implementation of matrix multiplication in LabVIEW does not use data parallelism.

Data parallelism; by dividing the matrix in half the operation can be computed simultaneously on two CPU cores.

With 1000 x 1000 matrices used for the input matrices:

<table>
<thead>
<tr>
<th></th>
<th>Execution Time on Single Core Processor</th>
<th>Execution Time on Dual Core Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication without Data Parallelism</td>
<td>1.195 seconds</td>
<td>1.159 seconds</td>
</tr>
<tr>
<td>Matrix Multiplication with Data Parallelism</td>
<td>1.224 seconds</td>
<td>0.629 seconds</td>
</tr>
</tbody>
</table>
Data Flow

• Many applications involve sequential, multistep algorithms
The five tasks run in the same thread because they are connected sequentially.
Data Flow Parallelism - Pipelining

- Applying pipelining can increase performance
  - Increase throughput (amount of data processed in a given time period)
- Pipelining Strategy:
Pipelining in LabVIEW

Note: Queues may also be used to pipeline data between different loops
In general, the CPU and data bus operate most efficiently when processing large blocks of data.

Note: Pipelined processing does introduce latency between input and output!
Pipelining increases latency

- Pipelining increases throughput but it also introduces additional latency.
Multicore Programming Challenges

- Thread Synchronization
- Race Conditions
- Deadlocks
- Shared resources
- Data transfer between processor cores
Synchronization in LabVIEW

• Synchronization mechanisms in LabVIEW:
  – Notifiers
  – Queues
  – Semaphores
  – Rendezvous
  – Occurrences
Data Transfer between cores

- Physical distance between processors and the quality of the processor connections can have a large effect on execution speed.
Unbalanced vs. balanced Parallel Tasks

Unbalanced

Balanced
Pipelining and balancing

- In order to gain the most performance increase possible from pipelining, individual stages must be carefully balanced so that no single stage takes a much longer time to complete than other stages.
- In addition, any data transfer between pipeline stages should be minimized to avoid decreased performance due to memory access from multiple cores.
LabVIEW - Parallel For-loops

- Enabling of iteration parallelism on For Loops.
- This feature can be applied to a For Loop if the computation in one iteration does not depend on the results from another iteration.
- With iteration parallelism enabled, the iterations of the loop execute in parallel on multiple cores.
LabVIEW - Parallel For-loops II

- right-click on a For Loop, select Configure Iteration Parallelism..., and check Enable loop iteration parallelism in the dialog box
- **Number of generated parallel loop instances** is the maximum amount of parallelism you expect to need for this loop
- The value you wire into the parallel instances (P) terminal of your For Loop is the number of parallel instances you want to use at run-time (e.g. fewer cores than available)
Multicore options for VISION in LabVIEW

- **Requires:** NI Vision Development Module
- Sets the number of available processor cores to use for NI Vision applications

<table>
<thead>
<tr>
<th>Get/Set Number of Cores (Get)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get (0)</td>
<td>(Default) Returns the number of available processor cores</td>
</tr>
<tr>
<td>Set (1)</td>
<td>Specifies the number of processor cores available to NI Vision</td>
</tr>
<tr>
<td>Set Max Available (2)</td>
<td>Specifies that NI Vision should have access to all available processor cores</td>
</tr>
</tbody>
</table>

**Get/Set Number of Cores (Get)** specifies whether the VI gets or sets the number of processor cores available to NI Vision.

**Specified Number of Cores** specifies the number of processor cores available to NI Vision.
Explicit Threading with Timed Structures

- **Timed**(while) loop used in **real-time** (embedded) systems

- Code within timed structures will execute in precisely 1 thread (no more)
- Can be assigned a relative **priority**
- Can be used to set processor affinity
Timed loop in LabVIEW

- When code that is contained in a Timed Loop is run on a dual-core or multicore system, a unique thread will be created. The figure below demonstrates how two Timed Loops will create two unique threads which can be balanced across two separate cores of a multicore system.
Conclusions

- PC-based instrumentation benefit greatly from advances in multicore processor technology and improved data bus speeds.
- As new CPUs improve performance by adding multiple processing cores, parallel or pipelined processing structures are necessary to maximize CPU efficiency.
- Fortunately, LabVIEW solves this programming challenge by dynamically assigning processing tasks to individual processing cores. As illustrated, you can achieve significant performance improvements by structuring LabVIEW algorithms to take advantage of parallel processing.
Multicore Resources for LabVIEW

- www.ni.com/multicore