Trends in Parallel Programming
The Emergence of OpenMP

Presented to:
EECC 756
May 16, 2006
Trends in Parallel Programming
(Outline)

• Question: What drives parallel programming?
  – Need → Successful Research: Industry Adoption
  – Low Cost, Available Hardware:
    • Single/Multiple-CPU Workstations and Fast Network Interconnections.

• Parallel Programming Trends: History and Introduction
  – MPI, PVM – Message Passing Models work in most parallel computing systems.
    • Is there a need for Declarative-Based Parallel Programming??
  – X3H5, OpenMP – Thread-Based Parallelism works VERY well in Shared-Memory computing systems.

• OpenMP:
  – Design Objectives
    • Ease of use: Message Passing is hard.
    • Control Structures, Data Environment, Synchronization, Run-Time Library, etc..
  – OpenMP Programming Model
    • Fork-Join Thread Programming.
    • Same Sequential/Parallel Program – with or without Declarations Processed.
  – OpenMP Program Example/Syntax
    • Sequential / Parallel Summation.
    • Illustration of Execution in Shared Address Space
  – Analysis of OpenMP
    • Advantages/Disadvantages.

• Parallel Programming Trends: Hybrid Models and Reflection
  – SALIENT POINT:
    • A hybrid approach to parallel computing is the most obvious, with a blend of message passing and compiler declarations assisting in thread manipulation/concurrency.
What influences the trends?

• Industry/Research
  – Adoption of standards: Cooperation within Industry.
  – Portability: Parallel Programming Development

• Hardware
  – Shared-memory multiprocessors are becoming more popular.
  – Availability of networks with high bandwidth/low latency.
  – Realistic scalability in terms of (cost $ and space)/node.
Trends: Industry Influences

• Adoption of Languages/Technology Standards
  – Standards put technology on a fast-track toward industry wide acceptance and development.

• Portability
  – New technologies should be backward compatible.
  – Re-use of current code/technology should be exploited.

• Efficiency
  – New technology is entertained if monetary cost and time cost is effective.
    • e.g. - Updating older serial code to exploit new hardware.
Trends: Hardware Influences

- Typical Cluster of Workstation utilizing current parallel programming technology: MPI, PVM, etc..

- Emerging Clusters still utilizing current parallel programming technology: MPI, PVM, etc..

- Message Passing over network interconnect is adequate:
Parallel Programming Trends vs. History

**Distributed Computing:**
- PVM: (1989-present)
  - Oak Ridge National Laboratory development for heterogeneous cluster computing
  - Several re-writes and versions are available.
- MPI: (1994-present)
  - De Facto Standard of Distributed Computing Message Passing.
  - Industry and Research Groups: consolidated effort to standardize message passing.

**Shared-Memory Multiprocessor Computing:**
  - Informal industry effort to standardize directives with FORTRAN77
- ANSI X3H5: (1994)
  - Formal PCF effort: Implementations were all similar, but diverging.
  - REJECTED: Distributed Memory system were becoming popular.

**WHY USE DIRECTIVE BASED?? PVM, MPI can be used. ***

- OpenMP: (1997-present)
  - Standards Published – Review Boards, both private and public.
  - Becoming the De Facto Standard of Shared-Memory Computing.
- Open specifications for (M)ulti(P)rocessing: ver. 2.5 (2005).
  - A portable and scalable API for multi-platform shared memory multiprocessing (SMM) programming in any language.
    - compiler directives / library routines / environment variables.
    - C/C++ and FORTRAN77/90 compilers have been extended to support OpenMP.
  - Designed and targeted for programmers who want to easily parallelize older serial code
    - Sequential Code $\rightarrow$ Parallel Code Time: a major consideration.
  - Defined and developed by OpenMP architecture review board (OARB) group.
    - Consists of representatives from major computer hardware and software vendors.
OpenMP Design Objectives

• Control Structure
  – Parallel sections
  – Sequential sections
  – Private and Shared Data and parallel regions.

• Data Environment
  – Shared and private data variables.

• Synchronization Primitives
  – Barriers
  – Critical Sections / Atomic Sections
  – Locks (Semaphores/Mutex)

• Run-time Library
  – Functions include, but not limited to
    • modify/check the number of threads.

• Environment Variables
  – e.g., default number of threads in a system: matching number of local CPU in Node.
OpenMP Programming Model

• **Fork-Join programming model:**
  – A program begins and exits with a single thread of execution.
  – Parallel constructs are separate threads spawned as a team and consolidated by the master thread.

• **Absence of real-time checking:**
  – Task Dependencies.
  – Deadlocks/Race-Conditions.
  – Other problems that result in incorrect program execution.

• **Sequential and Parallel Program:**
  – There is only one version of the code; OpenMP declarations are either processed or ignored.

• **OpenMP requires Thread Safety:**
  – All standard C built-in functions used are required to be thread-safe.
OpenMP Syntax / Example

#pragma omp parallel (for/do) [clause …] newline
  if( scalar expression )
  num_threads( scalar )
  private( list )
  public( list )
  firstprivate ( list ) /* data is private, but init from master */
  lastprivate ( list ) /* data is private, last copied to memory */
  reduction ( operator: list ) /* consolidation */
  copyin ( list ) /* argument, passed from global memory */

#pragma omp section
  critical, barrier, atomic, flush, lock
  master, single
  threadprivate
#include <omp.h>
#define MAX_NUM (15)
int main( int argc, char *argv[ ] ) {

    int sum = 0;
    int  i = 0;

    for ( i = 0; i < MAX_NUM; i++ ){  /** EXPLOIT PARALLEL NATURE OF LOOP**/
        sum += i;
    }

} /* main */
OpenMP within Sequential Program Example

```c
#include <omp.h>
#define MAX_NUM (15)
int main (int argc, char *argv[]) {
    int sum = 0;
    int i = 0;
    #pragma omp parallel do num_threads(4) private(i) reduction(+:sum) {
        #pragma omp single {
            printf(“Hello from thread team! I am thread %d\n”, omp_get_thread_num( ));
        }
        for ( i = 0 ; i < MAX_NUM ; i++ ) {
            sum += i;
        }
    }
} /* main */
```
OpenMP Example: Execution Illustration

Thread 0 (Master Thread)

- `i = 0, 1, 2, 3`
- `sum = 0;`
- `i = 0;`
- `printf(...) call`
- `0+1+2+3 = 6`

Thread 0

- `i = 4, 5, 6, 7`
- `sum = 4+5+6+7 = 22`
- `i = 4, 5, 6, 7`
- `sum = 4+5+6+7 = 22`

Thread 1

- `i = 8, 9, 10, 11`
- `sum = 8+9+10+11 = 38`

Thread 2

- `i = 12, 13, 14, 15`
- `sum = 12+13+14+15 = 54`

Thread 3

- `i = 12, 13, 14, 15`
- `sum = 12+13+14+15 = 54`

Shared Memory

- `sum = 6+22+38+54`
- `i = 0;`

- `shared memory write to [i]=[sum]=0`
- `shared memory write to [sum] = 120`
- `implied Barrier`
<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simple: No overhead as with message passing interfaces; data is accessible by all threads.</td>
<td>1. Currently only runs efficiently in shared-memory multiprocessor platforms. Does not support distributed shared memory (DSM) based on non Non-Uniform Memory Access (NUMA) architecture.</td>
</tr>
<tr>
<td>2. Data access and layout handled automatically by compiler directives.</td>
<td>2. Requires a compiler that supports OpenMP declarations.</td>
</tr>
<tr>
<td>3. One code package for serial and parallel code.</td>
<td>3. Low parallel efficiency. Leaves out a relatively high percentage of a non-loop code in sequential part.</td>
</tr>
<tr>
<td>4. Fork-Join model provides various grain sizes of parallelism.</td>
<td>4. There is not support for fast synchronization/low-latency communication between threads before and after runtime.</td>
</tr>
</tbody>
</table>
Trends: Programming Hybrid Models

*MPI and OpenMP inherently do not conflict.
Can be used together in emerging clusters better
than using solely MPI or OpenMP.
Reflection on Programming Trends

• **What influences parallel programming development?**
  – Hardware: Cheap, Available and Fast.
  – Industry: Ease of Use and Low Cost of Development

• **Do the current parallel languages satisfy current needs?**
  – Distributed Computing: MPI, PVM, etc.
    • Yes, but poor fit when Shared-Memory Nodes within cluster.
  – Shared Memory Computing: OpenMP.
    • Yes, but poor fit when used in Distributed Systems.

*** SALIENT POINT ***
A blend of languages that support all types of networked computing machines must be exploited to its full potential in the future.
References


Authors: Ryan Belanger (rcb8281@rit.edu)
Omer Farooq (oxf8548@rit.edu)