



## **Multicore Programming Practices: Defining Industry-wide, Best Practices to Leverage Existing Code in Multicore Environments**

**August 25, 2009**

**Presented by:**

**Max Domeika & David Stewart  
Co-Chairs of MPP Working Group**

# Presenters' Backgrounds

## ► David Stewart

- Co-founder and CEO of CriticalBlue
- Founder and Business Development Director of the Cadence System-on-Chip Product Design facility at the Alba Campus
- Co-Chair of MPP working group



## ► Max Domeika

- Intel Corporation – Technologist on Embedded, Multicore & Software Tools
- Author of “Software Development for Embedded Multi-core Systems”
- Co-Chair of MPP working group



- ▶ Overview of Multicore Association working groups
- ▶ Background on multicore programming
- ▶ Motivation for the creation of the MPP working group
- ▶ Logistics, Progress and Status
- ▶ Questions & Answers

## ► Currently Active

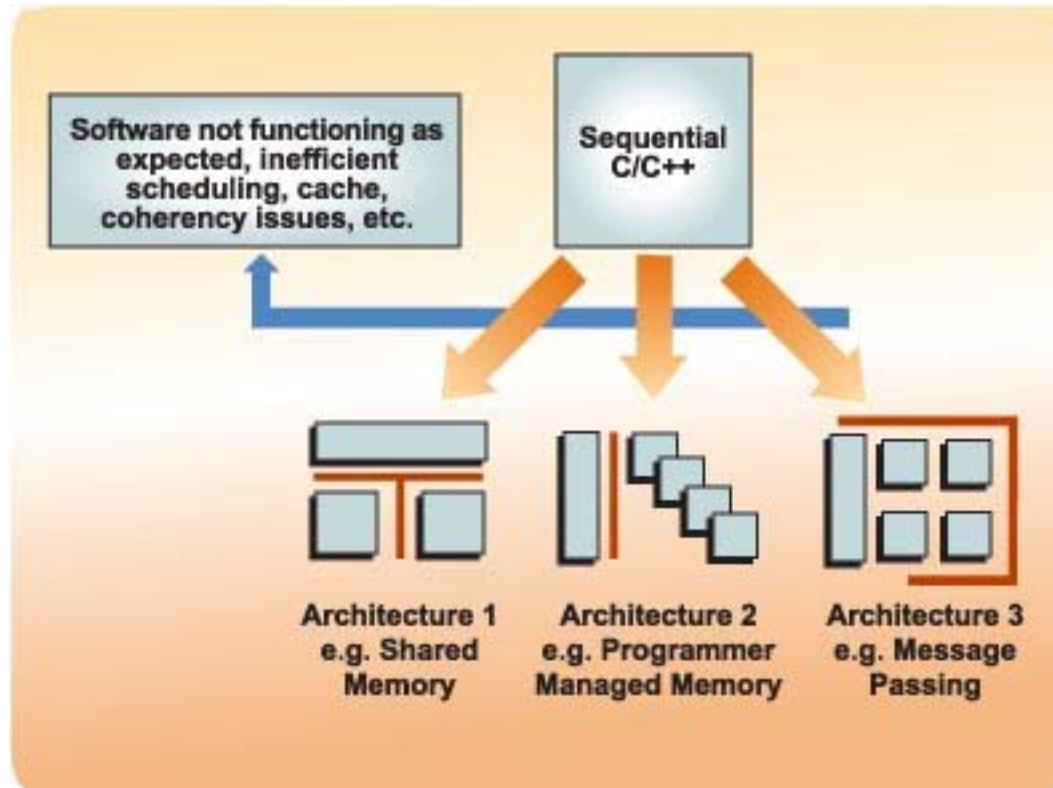
- Multicore Resource Management API (MRAPI)
- Multicore Programming Practices (MPP)
- Multicore Virtualization Working Group (MVWG)

## ► Completed

- Multicore Communications API (MCAPI)
- Downloaded now by almost 500 people
- Working its way into commercial products

# Facing General Programming Issues

- Fact: C/C++ will be predominant programming language for at least 8 years



- Problem: While we wait for long term research results, the multicore programmability gap is opening rapidly

# What Does The Industry Need Right Now?

- ▶ Continue with long term research into languages, methodologies, etc
- ▶ Short term direction as to how today's embedded C/C++ code may be written to be “multicore ready” today
- ▶ Influence of a group of like-minded methodology experts to ensure completeness, usefulness and industry-wide compatibility
- ▶ The creation of a standard “best practices” guide through a recognized, neutral industry body, based on capturing current best practices

# Action: Multicore Programming Practices Working Group

- ▶ Best practices for writing multicore-ready software using C/C++ without extensions
- ▶ Allow embedded software to be more easily compiled across a range of multicore processor platforms
- ▶ Framework of common pitfalls when transitioning from serial to parallel
- ▶ Consider solutions or avoidance tactics
- ▶ Minimize debugging efforts by reducing bugs

## Multicore Programming Practices (MPP)

The creation of MPP, a best practices guide to the writing of C/C++ embedded software, such that it may be more easily compiled across a range of multicore processor platforms.

MPP will be an open document, possibly a book or booklet, created by a working group operating under the Multicore Association standards body, and constructed in layers such that initial works may be delivered quickly, while being further refined. The document could also form the basis of future Association standards.

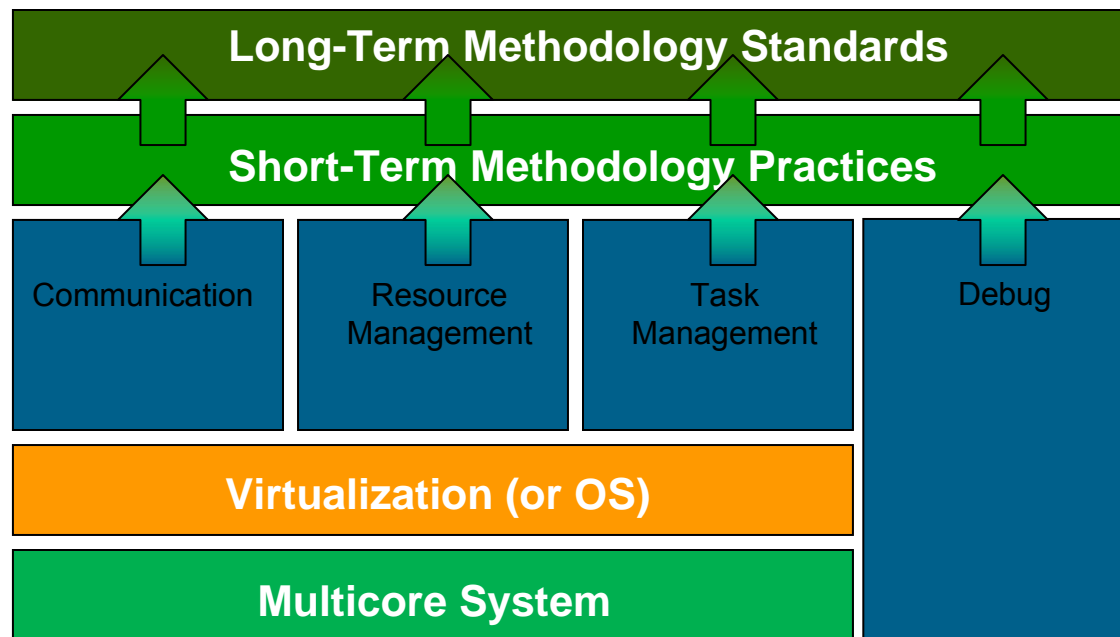
# Why The Multicore Association?

- ▶ The program should be conducted by an independent body, set up to deliberate on standards and cross industry practices
- ▶ The Multicore Association's charter is to focus on these type of industry issues
- ▶ The Association's membership has just the right expertise from member companies concerned about this issue and who would benefit from its results
  - Intel, Freescale, TI, WindRiver, Nokia Siemens Networks, Mentor Graphics, etc.



# Complements The MCA Strategy

- ▶ MPP can make use of MCA Pillars
  - Demonstrates their use in practical scenarios
- ▶ MPP can provide insights for standards programs
  - Allows issues to be considered for longer term projects



# What's in it for the Stakeholders

## ► Software Engineers

- Identification of issues to avoid
- Portability between platforms
- Improved software performance
- More predictable schedule

## ► Processor Providers

- Faster ramp up time for new software projects
- Better use of processor resources
- Easier programmability, lower support
- Recognition of processor usage intricacies

## ► Tool / Methodology Providers

- Increased focus for tool development
- Commonality between processors, reduced custom tool work



## ▶ Group started with charter

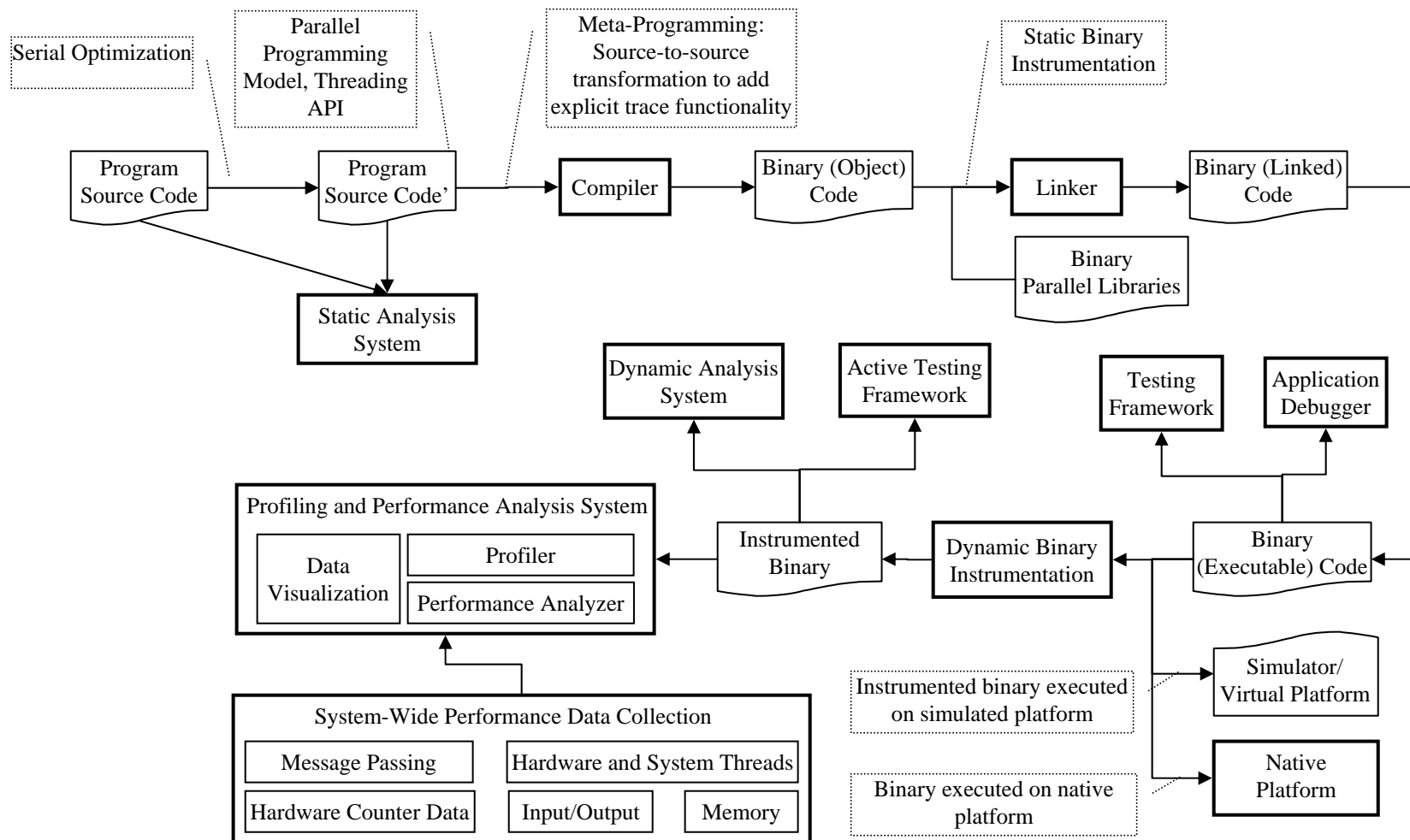
*The MPP guide details a concise set of best practices in software development for multicore processors using existing technology and existing software.*

- ▶ Approved chapter abstracts (August 2008)
- ▶ Approved Detailed outline (February 2009)
- ▶ Approved Chapter 1 (March 2009)
- ▶ Approved Chapter 2 (June 2009)
- ▶ Started writing 2 chapters (March - October 2009)
- ▶ Write last 2 chapters (post October 2009)

1. Introduction (8)
2. Overview of Available Technology (5)
3. Analysis and High Level Design (20)
4. Implementation and Low Level Design (15)
5. Debug (15)
6. Performance Tuning (15)
7. Glossary (4)

- ▶ Focus is on existing code, not new programming technology
- ▶ C/C++ language with no extensions
- ▶ Platforms prioritized as follows:
  - Homogenous multicore processor with shared memory using PThreads
  - Heterogeneous multicore processor with a mix of shared and non-shared memory using MCA MCAPI

# Multicore Tool Workflow



Brutch, Tasneem, "Parallel Programming Development Life Cycle: Understanding Tools and their Workflow when Migrating Sequential Applications to Multicore Platforms," USENIX ;login:, Oct. 2009.

## ► Implementation and Low Level Design (15 page budget)

### • Thread Based Implementations

- Dealing with Thread Safety
- Implementing Synchronizations and Locks
  - Barriers, Mutex, Nested Locks, Mutual exclusion, ...
  - Shared Data Implementation
  - Shared Queue Implementation
  - Granularity – discuss granularity of synchronization including lock-free, course-grained and fine-grained approaches
- Implementing Task Parallelism
  - Fork and Join
  - Parallel Pipeline Computation
  - Master/Worker Scheme
  - Divide and Conquer Scheme
  - Task Scheduling Considerations
  - Thread Pooling
  - Affinity scheduling, ...
- Event Based Parallel Programs
- Implementing Loop Parallelism
  - Aligning computation and locality
- Using Parallel Collective Operation

### • Message Passing Based Implementations

- Message Passing Communication
- Implementing Synchronizations and Locks
  - Barriers, ...
- Task Decomposition Model
- Data Decomposition Based Implementation
  - Distributed Arrays
- Using Parallel Collective Operation

### • Hybrid Implementations

**3.2.5.5 Data Movement** – different decompositions will require different amounts of data copying or communication. Different architectures will have different costs of communication (on-chip versus off-chip, uniform vs. NUMA, etc., data format differences) which will complicate data placement. Recommended practice- partition at as high a level as possible; check hot spot coverage, synchronization, and communication costs. Hot Spot example – a hot spot routine which is a high percentage of the run time in total but which is called many times is too fine grained. Move up the call tree to reduce the synchronization overhead.

### **3.2.6 Load Balancing** – discuss static and dynamic workloads

Static workload – give example of simple static loop with a fixed amount of work per iteration- split load into fixed amounts of work.

Dynamic workload – modify example to have dynamic work per iteration. Discuss inefficiency of fixed number of iterations (static scheduling) - discuss finer-grained dynamic scheduling to improve workload balance.

### **3.2.7 Choice of Algorithm**

Some algorithms are more amenable to parallelism than others; when dependencies limit parallelism, consider alternative approaches.

### **3.2.8 Design Patterns** – show some representative design patterns, for example:

Master-Workers – common data decomposition pattern; include basic fork-join approach.

Reduction – show how to refactor a loop with an associative reduction calculation.

## 1. Gotchas: Avoiding False Sharing

Shared memory systems are architected with multiple processors sharing one unified memory system. Each processor has its own local cache to reduce the apparent memory latency.

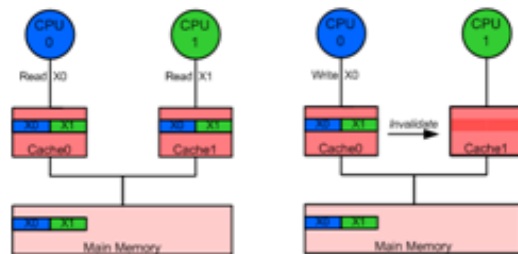
Multiple processors will compete for access to main memory, and may read and write to the same locations. Cache coherency schemes must be used to guarantee that all processors have a consistent view of main memory. When one processor writes a memory location cached by other processors, the other cached copies must be updated or invalidated, which reduces memory performance.

### 1.1. False Sharing

This coherency penalty may also occur for independent memory values. For example, assume that two processors read contiguous data values; CPU0 reads `x0` while CPU1 reads `x1`.

Each CPU reads its value into its own cache. When the main memory is read, a cache line's worth of data, typically 32 bytes, is read into the cache. Since the values are contiguous, it's likely that each CPU read both values into its cache line.

Even though they don't need both values, the cache lines now share the same data. If CPU0 then modifies `x0` and writes it, the cache line in CPU1 must be invalidated while the CPU0's cache line is written back to main memory. When CPU1 needs value `x1`, it will need to reread it from main memory, even though the actual value has not changed.



This situation is called false sharing and occurs because of poor data structure layout or interleaved data access patterns between threads.

### 1.2. Poor Memory Access Pattern

A simple example using `memcpy` demonstrates a poor access pattern:

```
typedef struct {
    int32_t *arr; // array origin
    int32_t n; // array size
    int32_t i0; // starting index
    int32_t di; // index step per iteration
};
```

```
void compute_args_t;

void *compute_partition(void *vargs) {
    uint32_t i;
    compute_args_t *args = (compute_args_t *) vargs;
    int32_t *px = args->px;
    uint32_t n = args->n;
    uint32_t i0 = args->i0;
    uint32_t di = args->di;

    for (i = i0; i < n; i += di) {
        *(px + i) = correct(*(px + i));
    }

    memcpy(&args->data, NULL, sizeof(args->data));
}

void compute_0000000000(int32_t x[], uint32_t n) {
    memcpy(&args->threads, NULL, sizeof(args->threads));
    memcpy(&args->args, NULL, sizeof(args->args));
    void *status;
    int m, t;
    // launch worker threads
    for (t = 0; t < NUM_THREADS; ++t) {
        args[t].px = x;
        args[t].n = n;
        args[t].i0 = t;
        args[t].di = NUM_THREADS;
        pthread_create(&threads[t], NULL, compute_0000000000, &args[t]);
    }

    // wait for worker threads to complete
    for (t = 0; t < NUM_THREADS; ++t) {
        pthread_join(threads[t], &status);
    }
}
```

In this example, a large array of values is adjusted by a simple function, `correct()`. The example is embarrassingly parallel since there are no data dependencies between the array values. The data may be partitioned in any way between threads to do the corrections in parallel. Unfortunately, the programmer has chosen to have the threads access the data array in a tightly interleaved pattern. The processors are filling their caches with the same data, and each data cache line contains only a fraction,  $1/\text{NUM\_PROCESSORS}$ , of usable data values. Worse, updating each data value in one cache line invalidates all the ones. The false sharing greatly diminishes the algorithm performance.

### 1.3. Memory Access Optimization

A far better approach is to break the data array into chunks of contiguous data values.

```
typedef struct {
    int32_t *arr; // array origin
    uint32_t i0; // starting index
    int32_t di; // chunk size
    int32_t di; // index step per iteration
};
```

## 3. Avoiding False Sharing

Shared memory systems are architected with multiple processors sharing one unified memory system. Each processor has its own local cache to reduce the apparent memory latency. Multiple processors will compete for access to main memory, and may read and write to the same locations. Cache coherency schemes must be used to guarantee that all processors have a consistent view of main memory. When one processor writes to a memory location cached by other processors, the other cached copies must be updated or invalidated, which reduces memory performance.

### 3.1. False Sharing

This coherency penalty may also occur for independent memory values. For example, assume that two processors read contiguous data values; cpu0 reads x0 while cpu1 reads x1. Each CPU reads its value into its own cache. When the main memory is read, a cache line's worth of data, typically 32 bytes, is read into the cache. Since the values are contiguous, it is likely that each CPU read 'both' values into its cache line. Even though they don't need both values, the cache lines now share the same data. If cpu0 then modifies x0 and writes it, the cache line in cpu1 must be invalidated while the cpu0 cache line is written back to main memory. When cpu1 needs value x1, it will need to reread it from main memory, even though the actual value has not changed.

# Working Group Logistics

- ▶ Phone meetings every 2 weeks
- ▶ 1 hour duration maximum
- ▶ Review of action items, issue discussion, assignments for the next period
- ▶ Chapter writing in smaller groups
- ▶ Chapter review by the whole group

# Benefits to Participating

- ▶ Be sure that you understand what the industry standards and best known methods are
- ▶ Gain experience and be better prepared to migrate to the multicore programming world
- ▶ Debate ideas with key industry participants
- ▶ Be part of the industry standard 'Read Me' guide for Multicore Software programming

- ▶ Problem: While we are waiting for long term research results, the multicore programmability gap is opening rapidly
- ▶ Action: MPP Working Group – Best practices for writing multicore-ready embedded software
- ▶ Objective: Meet immediate needs of multicore stakeholders in an open, efficient and effective manner

# Multicore Association Membership

## ▶ Executive Board Membership

- Freescale Semiconductor, Huawei Technologies, Intel, Mentor Graphics, Nokia Siemens Networks, Plurality, PolyCore Software, Samsung Electronics, Texas Instruments, Tiler, Wind River

## ▶ Working Group Membership

- CAPS entreprise, Codeplay, CriticalBlue, Enea, eSOL, IMEC, LG Electronics, LSI, MIPS Technologies, National Instruments, nCore Design, Open Kernel Labs, Virtutech, VMware

## ▶ University Membership

- Carnegie Mellon University, Delft University of Technology, University of Utah

**Become a Member of The Multicore Association Today -  
Don't Let the Multicore Revolution Leave You Behind.**

**[www.multicore-association.org](http://www.multicore-association.org)**

# Questions and Answers

**Asymmetric Multiprocessing (AMP) Multi-core or multiprocessor**

configuration where two or more CPUs have a different and in many cases disjoint view of the system.

**Automatic Vectorization Compiler** optimization that changes code that computes individually on scalar data elements to use instructions that compute on multiple data elements at the same time.

**Cache Thrashing Cache** access behavior that results in data being brought into and flushed out of cache in quick succession typically resulting in poor performance.

**Coarse Grain Locking Concurrency** control technique that employs synchronization around data elements where only some of the data elements have the potential to cause a threading issue. This approach may sacrifice performance as some data accesses are unnecessarily synchronized, but is typically easier to code than fine grain locking.

## ► Implementation and Low Level Design (15 page budget)

### • Thread Based Implementations

- Dealing with Thread Safety
- Implementing Synchronizations and Locks
  - Barriers, Mutex, Nested Locks, Mutual exclusion, ...
  - Shared Data Implementation
  - Shared Queue Implementation
  - Granularity – discuss granularity of synchronization including lock-free, course-grained and fine-grained approaches
- Implementing Task Parallelism
  - Fork and Join
  - .....
  - .....