Heterogeneous Computing (HC) & Micro-Heterogeneous Computing (MHC)

- High Performance Computing (HPC) Trends
- Steps in Creating a Parallel Program
  - Factors Affecting Parallel System Performance
- Scalable Distributed Memory Processors: MPPs & Clusters
- Limitations of Computational-mode Homogeneity in Parallel Architectures
- Heterogeneous Computing (HC)
- Proposed Computing Paradigm: Micro-Heterogeneous Computing (MHC)
  - Framework of Proposed MHC Architecture
  - Design Considerations of MHC-API
  - Analytical Benchmarking & Code-Type Profiling in MHC
  - Formulation of Mapping Heuristics For MHC
  - Example Initial MHC Scheduling Heuristics Developed
  - Modeling/Simulation of MHC Framework & Preliminary Results
  - MHC Framework Linux Implementation & Preliminary Results
- Future Work in MHC.

OpenCL
High Performance Computing (HPC) Trends

- Demands of engineering and scientific applications is growing in terms of:
  - Computational and memory requirements
  - Diversity of computation modes present.
- Such demands can only be met efficiently by using large-scale parallel systems that utilize a large number of high-performance processors with additional diverse modes of computations supported.
- The increased utilization of commodity of-the-shelf (COTS) components in high performance computing systems instead of costly custom components.
  - Commercial microprocessors with their increased performance and low cost almost replaced custom processors used in traditional supercomputers.
- Cluster computing that entirely uses COTS components and royalty-free software is gaining popularity as a cost-effective high performance alternative to commercial “Big-Iron” solutions: Commodity Supercomputing.
- The future of high performance computing relies on the efficient use of clusters with symmetric multiprocessor (SMP) nodes and scalable interconnection networks.
- General Purpose Processors (GPPs) used in such clusters are not suitable for computations that have diverse and specialized computational mode requirements such as digital signal processing, logic-intensive, or data parallel computations. Such applications, when run on clusters, currently only achieve a small fraction of the potential peak performance.
High Performance Computing (HPC)  
Application Areas

- Astrophysics
- Atmospheric and Ocean Modeling
- Bioinformatics
- Biomolecular simulation: Protein folding
- Computational Chemistry
- Computational Fluid Dynamics
- Computational Physics
- Computer vision and image understanding
- Data Mining and Data-intensive Computing
- Engineering analysis (CAD/CAM)
- Global climate modeling and forecasting
- Material Sciences
- Military applications
- Quantum chemistry
- VLSI design
- .....

From 756
Scientific Computing Demands

Demands exceed the capabilities of even the fastest current uniprocessor systems

Grand Challenge problems
- Global change
- Human genome
- Fluid turbulence
- Vehicle dynamics
- Ocean circulation
- Viscous fluid dynamics
- Superconductor modeling
- Quantum chromo dynamics
- Vision

From 756

Computational performance requirement

Storage requirement

(Memory Requirement)

1 TB

100 GB

10 GB

1 GB

100 MB

10 MB

2D airfoil

Oil reservoir modeling

48-hour weather

3D plasma modeling

72-hour weather

Vehicle signature

Structural biology

Pharmaceutical design

Chemical dynamics

From 756
LINPAK Performance Trends

Uniprocessor Performance

Parallel System Performance

From 756
Peak FP Performance Trends

- Petaflop: $10^{15}$ FLOPS = 1000 TFLOPS
- Teraflop: $10^{12}$ FLOPS = 1000 GFLOPS

From 756
Steps in Creating Parallel Programs

Partitioning

Sequential computation → Tasks → Processes → Parallel program → Processors

- 4 steps:
  - Decomposition, Assignment, Orchestration, Mapping
    - Done by programmer or system software (compiler, runtime, ...)

From 756
Levels of Parallelism in Program Execution

Level 5
- Jobs or programs (Multiprogramming)

Level 4
- Subprograms, job steps or related parts of a program

Level 3
- Procedures, subroutines, or co-routines

Level 2
- Non-recursive loops or unfolded iterations

Level 1
- Instructions or statements

Increasing communications demand and mapping/scheduling overhead

Coarse Grain
- Medium Grain
- Fine Grain

Higher degree of Parallelism

From 756
Parallel Program Performance Goal

- Parallel processing goal is to maximize speedup:

\[
\text{Speedup} = \frac{\text{Time}(1)}{\text{Time}(p)} \leq \frac{\text{Sequential Work on one processor}}{\text{Max} (\text{Work + Synch Wait Time + Comm Cost + Extra Work})}
\]

- Ideal Speedup = \( p \) = number of processors

- By:
  1. Balancing computations on processors (every processor does the same amount of work).
  2. Minimizing communication cost and other overheads associated with each step of parallel program creation and execution.

- Performance Scalability:

  Achieve a good speedup for the parallel application on the parallel architecture as problem size and machine size (number of processors) are increased.

From 756
Factors Affecting Parallel System Performance

• Parallel Algorithm-related:
  – Available concurrency and profile, grain, uniformity, patterns.
  – Required communication/synchronization, uniformity and patterns.
  – Data size requirements.
  – Communication to computation ratio.

• Parallel program related:
  – Programming model used.
  – Resulting data/code memory requirements, locality and working set characteristics.
  – Parallel task grain size.
  – Assignment/mapping: Dynamic or static.
  – Cost of communication/synchronization.

• Hardware/Architecture related:
  – Total CPU computational power available.
  – Types of computation modes supported.
  – Shared address space Vs. message passing.
  – Communication network characteristics (topology, bandwidth, latency)
  – Memory hierarchy properties.

From 756
Reality of Parallel Algorithm & Communication/Overheads Interaction

From 756

EECC722 - Shaaban
Classification of Parallel Architectures

- **Data Parallel/Single Instruction Multiple Data (SIMD):**
  - A single instruction manipulates many data items in parallel. Include:
    - **Array processors:** A large number of simple processing units, ranging from 1,024 to 16,384 that all may execute the same instruction on different data in lock-step fashion (Maspar, Thinking Machines CM-2).
    - **Traditional vector supercomputers:**
      - First class of supercomputers introduced in 1976 (Cray 1)
      - Dominant in high performance computing in the 70’s and 80’s
      - Current vector supercomputers: Cray SV1, Fujitsu VSX4, NEC SX-5
    - Suitable for fine grain data parallelism.
    - Parallel Programming: Data parallel programming model using vectorizing compilers, High Performance Fortran (HPF).
    - Very high cost due to utilizing custom processors and system components and low production volume.
      - Do not fit current incremental funding models.
    - Limited system scalability.
    - Short useful life span.

From 756
Classification of Parallel Architectures

• **Multiple Instruction Multiple Data (MIMD):**
  – These machines execute several instruction streams in parallel on different data.
  – **Shared Memory or Symmetric Memory Processors (SMPs):**
    • Systems with multiple tightly coupled CPUs (2-64) all of which share the same memory, system bus and address space.
    • Suitable for tasks with medium/coarse grain parallelism.
    • Parallel Programming: Shared address space multithreading using POSIX Threads (pthreads) or OpenMP.
  – **Scalable Distributed Memory Processors:**
    • Include commercial Massively Parallel Processor systems (MPPs) and computer clusters.
    • A large number of computing nodes (100s-1000s). Usually each node is a small scale (2-4 processor) SMPs connected using a scalable network.
    • Memory is distributed among the nodes. CPUs can only directly access local memory in their node.
    • Parallel Programming: Message passing over the network using Parallel Virtual Machine (PVM) or Message Passing Interface (MPI)
    • Suitable for large tasks with coarse grain parallelism and low communication to computation ratio.

From 756
Scalable Distributed Memory Processors: MPPs & Clusters

MPPs vs. Clusters
- **MPPs**: “Big Iron” machines
  - COTS components usually limited to using commercial processors.
  - High system cost
- **Clusters**: Commodity Supercomputing
  - COTS components used for all system components.
  - Lower cost than MPP solutions

Operating system?
MPPs: Proprietary
Clusters: royalty-free (Linux)

Communication Assist:
MPPs: Custom
Clusters: COTS

Distributed Memory

Node: O(10) SMP
MPPs: Custom node
Clusters: COTS node (workstations or PCs)

Custom-designed CPU?
MPPs: Custom or commodity
Clusters: commodity

Parallel Programming:
Between nodes: Message passing using PVM, MPI
In SMP nodes: Multithreading using Pthreads, OpenMP

Scalable Network:
- Low latency
- High bandwidth
MPPs: Custom
Clusters: COTS

- Gigabit Ethernet
- System Area Networks (SANS)
  - ATM
  - Myrinet
  - SCI

Include computing elements other than GPPs?
A Major Limitation of Homogeneous Supercomputing Systems

- Traditional homogeneous supercomputing system architectures usually support a single homogeneous mode of parallelism including: Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and vector processing.
- Such systems perform well when the application contains a single mode of parallelism that matches the mode supported by the system.
- In reality, many supercomputing applications have subtasks with different modes of parallelism.
- When such applications execute on a homogeneous system, the machine spends most of the time executing subtasks for which it is not well suited.
- The outcome is that only a small fraction of the peak performance of the machine is achieved.
- Image understanding is an example application that requires different types of parallelism.
Computational-mode Homogeneity of Cluster Nodes

• With the exception of amount of local memory, speed variations, and number of processors in each node, the computing nodes in a cluster are **homogeneous** in terms of computational modes supported.
  
  – This is due to the utilization of general-purpose processors (GPPs) that offer a single mode of computation as the only computing elements available to user programs.

• GPPs are designed to offer good performance for a wide variety of computations, but are not optimized for tasks with specialized and diverse computational requirements such as digital signal processing, logic-intensive, or data parallel computations.

• This limitation is similar to that in homogeneous supercomputing systems, and results in computer clusters achieving only a **small fraction** of their potential peak performance when running tasks that require different modes of computation.
  
  – This severely limits computing scalability for such applications.
Heterogeneous Computing (HC)

- Heterogeneous Computing (HC), addresses the issue of computational mode homogeneity in supercomputers by:
  - Effectively utilizing a heterogeneous suite of high-performance autonomous machines that differ in both speed and modes of parallelism supported to optimally meet the demands of large tasks with diverse computational requirements.
  - It is possible, through heuristics, to match a program with the architecture that best support it.
  - If a program is broken up, and each part performed on the best suited machine, the execution time can be greatly reduced.

Also known as: Network Heterogeneous Computing (NHC)

Heterogeneous Computing (HC) led to the current efforts in Grid Computing
Motivation For Heterogeneous Computing

- Hypothetical example of the advantage of using a heterogeneous suite of machines, where the heterogeneous suite time includes inter-machine communication overhead. Not drawn to scale.
Heterogeneous Computing Example Application: Image Understanding

- **Highest Level (Knowledge Processing):**
  - Uses the results from the lower levels to infer semantic attributes of an image
  - Requires coarse-grained loosely coupled MIMD machines.

- **Intermediate Level (Symbolic Processing):**
  - Grouping and organization of features extracted.
  - Communication is irregular, parallelism decreases as features are grouped.
  - Best suited to medium-grained MIMD machines.

- **Lowest Level (Sensory Processing):**
  - Consists of pixel-based operators and pixel subset operators such as edge detection
  - Highest amount of data parallelism
  - Best suited to mesh connected SIMD machines or other data parallel arch.

Paper HC-1
Heterogeneous Computing Broad Issues

- Analytical benchmarking
- Code-Type or task profiling
- Matching and Scheduling (mapping)
- Interconnection requirements
- Programming environments/APIs (e.g. PVM, MPI).
Steps of Applications Processing in Heterogeneous Computing (HC)

- **Analytical Benchmarking:**
  - This step provides a measure of how well a given machine is able to perform when running a certain type of code.
  - This is required in HC to determine which types of code should be mapped to which machines.

- **Code-type Profiling:**
  - Used to determine the type and fraction of processing modes that exist in each program segment.
  - Needed so that an attempt can be made to match each code segment with the most efficient machine.

- **Task Scheduling /Mapping:** tasks are mapped to a suitable machine using some form of mapping/scheduling heuristic

- **Task Execution:** the tasks are executed on the selected machine
Code-Type Profiling Example

- Example results from the code-type profiling of a task.
- The task is broken into $S$ segments, each of which contains embedded homogeneous parallelism.

Diagram:

- Code Segment 1: SIMD
- Code Segment 2: MIMD
- Code Segment S: Vector
HC Task Matching and Scheduling (Mapping)

- Task matching involves assigning a task to a suitable machine.
- Task scheduling on the assigned machine determines the order of execution of that task on the selected machine.
- The process of matching and scheduling tasks onto machines is called mapping.
- Goal of mapping is to maximize performance by assigning code-types to the best suited machine while taking into account the costs of the mapping including computation and communication costs based on information obtained from analytical benchmarking, code-type profiling and possibly system workload.
- The problem of finding optimal mapping has been shown in general to be NP-complete even for homogeneous environments.
- For this reason, the development of heuristic mapping and scheduling techniques that aim to achieve “good” sub-optimal mappings is an active area of research resulting in a large number of heuristic mapping algorithms for HC.
- Two different types of mapping heuristics for HC have been proposed, static or dynamic.

Paper HC-1, 2, 3, 4, 5, 6
HC Task Matching and Scheduling (Mapping)

Static Mapping Heuristics:

• Most such heuristic algorithms developed for HC are static and assume the ETC (expected time to compute) for every task on every machine to be known from code-type profiling and analytical benchmarking and not change at runtime.
  
• In addition many such heuristics assume large independent or meta-tasks that have no data dependencies.

• Even with these assumptions static heuristics have proven to be effective for many HC applications.

Dynamic Mapping Heuristics:

• Mapping is performed on-line (at runtime) taking into account current system workload.

• Research on this type of heuristics for HC is fairly recent and is motivated by utilizing the heterogeneous computing system for real-time applications.

Static: HC-3, 4, 5
Dynamic: HC-6

No account for system workload
Example Static Scheduling Heuristics for HC

• **Opportunistic Load Balancing (OLB):** assigns each task, in arbitrary order, to the next available machine.

• **User-Directed Assignment (UDA):** assigns each task, in arbitrary order, to the machine with the best expected execution time for the task.

• **Fast Greedy:** assigns each task, in arbitrary order, to the machine with the minimum completion time for that task.

• **Min-min:** the minimum completion time for each task is computed respect to all machines. The task with the overall minimum completion time is selected and assigned to the corresponding machine. The newly mapped task is removed, and the process repeats until all tasks are mapped.

• **Max-min:** The Max-min heuristic is very similar to the Min-min algorithm. The set of minimum completion times is calculated for every task. The task with overall maximum completion time from the set is selected and assigned to the corresponding machine.

• **Greedy or Duplex:** The Greedy heuristic is literally a combination of the Min-min and Max-min heuristics by using the better solution.
Example Static Scheduling Heuristics for HC

- **GA :** The Genetic algorithm (GA) is used for searching large solution space. It operates on a population of chromosomes for a given problem. The initial population is generated randomly. A chromosome could be generated by any other heuristic algorithm.

- **Simulated Annealing (SA):** an iterative technique that considers only one possible solution for each meta-task at a time. SA uses a procedure that probabilistically allows solution to be accepted to attempt to obtain a better search of the solution space based on a system temperature.

- **GSA :** The Genetic Simulated Annealing (GSA) heuristic is a combination of the GA and SA techniques.

- **Tabu :** Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas.

- **A* :** A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions and leaf nodes represent final solutions. Each node has a cost function, and the node with the minimum cost function is replaced by its children. Any time a node is added, the tree is pruned by deleting the node with the largest cost function. This process continues until a complete mapping (a leaf node) is reached.
Example Static Scheduling Heuristics for HC: The Segmented Min-Min Algorithm

• Every task has a ETC (expected time to compute) on a specific machine.
• If there are $t$ tasks and $m$ machines, we can obtain a $t \times m$ ETC matrix.
  – $ETC(i; j)$ is the estimated execution time for task $i$ on machine $j$.
• The Segmented min-min algorithm sorts the tasks according to ETCs.
• The tasks can be sorted into an ordered list by the average ETC, the minimum ETC, or the maximum ETC.
• Then, the task list is partitioned into segments with equal size.
• Each segment is scheduled in order using the standard Min-Min heuristic.
Segmented Min-Min Scheduling Heuristic

Segmented min-min (Smm)

1. Compute the sorting key for each task:

   Sub-Policy 1 - \textit{Smm-avg}: Compute the average value of each row in ETC matrix

   \[ key_i = \frac{\sum_j ETC(i, j)}{m} \]

   Sub-Policy 2 - \textit{Smm-min}: Compute the minimum value of each row in ETC matrix

   \[ key_i = \min_j ETC(i, j) \]

   Sub-Policy 3 - \textit{Smm-max}: Compute the maximum value of each row in ETC matrix

   \[ key_i = \max_j ETC(i, j) \]

2. Sort the tasks into a task list in decreasing order of their keys.
3. Partition the tasks evenly into \textit{N} segments.
4. Schedule each segment in order by applying Min-min.
Heterogeneous Computing System Interconnect Requirements

- In order to realize the performance improvements offered by heterogeneous computing, communication costs must be minimized.
- The interconnection medium must be able to provide high bandwidth (multiple gigabits per second per link) at a very low latency.
- It must also overcome current deficiencies such as the high overheads incurred during context switches, executing high-level protocols on each machine, or managing large amounts of packets.
- While the use of Ethernet-based LANs has become commonplace, these types of network interconnects are not well suited to heterogeneous supercomputers (high latency).
- This requirement of HC led to the development of cost-effective scalable system area networks (SANS) that provide the required high bandwidth, low latency, and low protocol overheads including Myrinet and Dolphin SCI interconnects.
- These system interconnects developed originally for HC, currently form the main interconnects in high performance cluster computing.
Development of Message Passing Environments/APIs for HC

- Since the suite of machines in a heterogeneous computing system are loosely coupled and do not share memory, communication between the cooperating subtasks must be achieved by exchanging messages over the network.

- This requirement led to the development of a number of platform-independent message-passing programming environments and APIs that provide the source-code portability across platforms.

- Parallel Virtual Machine (PVM), and Message Passing Interface (MPI) are the most widely-used of these environments.

- This also played a major role in making cluster computing a reality.
Heterogeneous Computing Limitations

• **Task Granularity**
  – Heterogeneous computing only utilizes **coarse-grained parallelism** to increase performance
  – Coarse-grained parallelism results in large task sizes and reduced coupling which allows the processing elements to work more efficiently
  – Requirement is also translated to most heterogeneous schedulers since they are based on the scheduling of meta-tasks, i.e. tasks that have no dependencies

• **Communication Overhead**
  – Tasks and their working sets must be transmitted over some form of network in order to execute them, latency and bandwidth become crucial factors
  – Overhead is also incurred when encoding and decoding the data for different architectures

• **Cost**
  – Machines used in heterogeneous computing environments can be prohibitively expensive
  – Expensive high speed, low latency networks are required to achieve the best performance
  – Not cost effective for applications where only a small portion of the code would benefit from such an environment
Computing System Element Choices

Programmability / Flexibility

GPPs
General Purpose Processors
Superscalar VLIW
DSPs
Network Processors
Graphics Processors
Application Specific Processors
Re-configurable Hardware
Co-Processors
ASICs

Selection Factors:
- Type and complexity of computational algorithm (general purpose vs. Specialized)
- Desired level of flexibility and programmability
- Performance requirements
- Desired level of computational efficiency
- Power requirements - Real-time constraints
- Development time and cost - System cost

Specialization, Development cost/time
Performance/Chip Area/Watt (Computational Efficiency)
Computing Element Choices Observation

- **Generality and efficiency are in some sense inversely related to one another:**
  - The more general-purpose a computing element is and thus the greater the number of tasks it can perform, the less efficient (e.g. Computations per chip area /watt) it will be in performing any of those specific tasks.
  - Design decisions are therefore almost always compromises; designers identify key features or requirements of applications that must be met and and make compromises on other less important features.

- **To counter the problem of computationally intense and specialized problems for which general purpose machines cannot achieve the necessary performance/other requirements:**
  - Special-purpose processors (or Application-Specific Processors, ASPs), attached processors, and coprocessors have been designed/built for many years, for specific application domains, such as image or digital signal processing (for which many of the computational tasks are specialized and can be very well defined).
Efficient Utilization of Chip Density Increase

• Increasing gap between "peak" performance of general-purpose processors and "average actually achieved" performance.
  – Most programmers don't write code that gets anywhere near the peak performance of current superscalar CPUs

• Increasing number of transistors on a (processor) chip (one billion+): How to use them efficiently?
  – Bigger caches (Most popular)?
  – Multiple processor cores? (Chip Multiprocessors - CMPs)
  – SMT support?
  – IRAM-style vector/memory?
  – DSP cores or other application specific processors?
  – Reconfigurable logic (FPGA or other reconfigurable logic)?

A Combination of the above choices?

Heterogeneous Computing System on a Chip?

Micro-Heterogeneous Computing (MHC)?
Micro-Heterogeneous Computing (MHC)

- Address computational-mode homogeneity in computer cluster nodes
- The utilization of faster microprocessors in cluster nodes cannot resolve this issue.
  - The development of faster GPPs only increases peak performance of cluster nodes without introducing the needed heterogeneity of computing modes.

- Micro-Heterogeneous Computing (MHC) is a computing paradigm proposed to address performance limitations resulting from computational-mode homogeneity in computing nodes by extending the benefits of heterogeneous computing to the single-node or chip level.
Micro-Heterogeneous Computing (MHC)

- Micro-Heterogeneous Computing (MHC) is defined as the efficient and user-code transparent utilization of heterogeneous modes of computation at the single-node or chip level. The GPP-based nodes are augmented with additional computing devices that offer different modes of computation.

- Devices that offer different modes of computation and thus can be utilized in MHC nodes include:
  - Digital Signal Processors (DSPs),
  - reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs),
  - Graphics Processor Units (GPUs),
  - vector co-processors and …
  - other future types of hardware-based devices that offer additional modes of computation.

- Such devices can be integrated into a base node creating an MHC node in three different ways: Levels of coupling or integration
  - Chip-level integration (on the same die as GPPs)
  - Integrated into the node at the system board-level. Or..
  - As COTS peripherals. Most cost-effective approach today

- In combination with base node GPPs, these elements create a small scale heterogeneous computing environment.
Micro-Heterogeneous Computing (MHC) Node: Levels of Coupling

Different levels of coupling between GPPs and other types of computing elements.

**Tight Coupling**
- Functional units (on chip)
- Coprocessor (on or off chip)

**Loose Coupling**
- Lower communication time/overheads (higher bandwidth/lower latency)
- External standalone processing unit (e.g. via network/IO interface)

MHC Node Diagram:

1. Coprocessor
2. Attached Processing Unit
3. Memory Caches
4. I/O Interface

Function Calls:
- Loose Coupling
- Tight Coupling

- Higher communication time/overheads (lower bandwidth/higher latency)
An example PCI(express)-based Micro-Heterogeneous (MHC) Node

One or more GPPs

Or:
PCI-X
PCI Express ...
HyperTransport ??

HC-7: Bill Scheidel’s MS Thesis

EECC722 - Shaaban
Scalable MHC Cluster Computing

- To maintain cost-effectiveness and scalability of computer clusters while alleviating node computational-mode homogeneity, clusters of MHC nodes are created by augmenting base cluster nodes with MHC PCI-based COTS.
Comparison Between Heterogeneous Computing and Micro-Heterogeneous Computing

- **Task Granularity**
  - Heterogeneous environments only support coarse-grained parallelism, while the MHC environment instead focuses on fine-grained parallelism by using a tightly coupled shared memory environment
  - Task size is reduced to a single function call in a MHC environment
  - Drawbacks:
    - Processing elements used in MHC are not nearly as powerful as the machines used in a standard heterogeneous environment
    - There is a small and finite number of processing elements that can be added to a single machine

- **Communication Overhead**
  - High performance I/O buses are twice as fast as the fastest network
  - Less overhead is incurred when encoding and decoding data since all processing elements use the same base architecture
  - Reduced + lower latency

- **Cost Effectiveness**
  - Machines used in a heterogeneous environment can cost tens of thousands of dollars each, and require the extra expense of the high-speed, low latency interconnects to achieve acceptable performance
  - MHC processing elements cost only hundreds of dollars
  - Improved

HC-7: Bill Scheidel’s MS Thesis
Possible COTS MHC Devices

- A large number of COTS add-on peripherals (e.g. PCI-based) are available that incorporate processing elements with desirable modes of computation (e.g. DSPs, FPGAs, vector processors).

- These devices are usually targeted for use in rapid system prototyping, product development, and real-time applications.

- While these devices are accessible to user programs, currently no industry standard device-independent Application Programming Interface (API) exists to allow device-independent user code access.

- Instead, multiple proprietary and device-specific APIs supplied by the manufacturers of the devices must be used.
  - This makes working with one of these devices difficult and working with combinations of devices quite a challenge.

This is also motivation behind recent development of OpenCL (Open Computing Language), a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors.

HC-7: Bill Scheidel’s MS Thesis

khronos.org
Example Possible MHC Devices

- **XP-15**
  - Developed by Texas Memory Systems
  - DSP based accelerator card
  - 8 GFLOPS peak performance for 32-bit floating point operations.
  - Contains 256 MB of on board DDR Ram
  - Supports over 500 different scientific functions
  - Increases FFT performance by 20x - 40x over a 1.4 Gigahertz Intel P4

- **Pegasus-2**
  - Developed by Catalina Research
  - Vector Processor based
  - Supports FFT, matrix and vector operations, convolutions, filters and more
  - Supported functions operate between 5x and 15x faster than a 1.4 Gigahertz Intel P4

HC-7: Bill Scheidel’s MS Thesis
Micro-Heterogeneous Computing (MHC): Challenges in Utilizing Diverse Computational Devices

- While a number of these COTS devices are good candidates for use in cost-effective MHC nodes, two very important issues must be addressed to successfully utilize them in an MHC environment:

  1. The use of proprietary APIs to access the devices is not user-code transparent:
     - Resulting code is tied to specific devices and is not portable to other nodes that do not have the exact configuration.
     - As a result, the utilization of these nodes in clusters is very difficult (MPI runs the same program on all cluster nodes), if not impossible.
     - Thus a device-independent API must be developed and supported by the operating system and target device drivers for efficient MHC node operation to provide the required device abstraction layer.

  2. In addition, the user is left to deal with the difficult issues of task mapping and load balancing, issues with which they may not be intimately familiar.
     - Thus operating system support for matching and scheduling (mapping) API calls to suitable devices must be provided for a successful MHC environment.
MHC Framework

• An OS-supported API-based backend designed to abstract the access to Heterogeneous processing elements in an MHC system.

• Motivation: High speed graphics did not come into their own until there were standard API’s for accessing them (e.g OpenGL, Direct-x).

• Main Framework Components:
  – A device-independent MHC-API in the form of dynamically-linked libraries to allow user-code transparent access to these devices must defined and implemented.
  
  – An efficient OS-supported dynamic matching and scheduling (mapping) mechanism to select the best suited device for a given MHC-API call and schedule it for execution to minimize execution time must be developed.
MHC Framework Requirements

• **Automatic, fully dynamic mapping**
  – Select devices, prepare code
  – Preserve effective execution order
  – Introduce parallel execution into single threaded programs.
  – Breaks programs up at the function call level

• **User transparency**
  – The program author does not have to know what devices it will be executed on.

• **Minimal user configuration required.**
Design Considerations of MHC-API

• Scientific APIs take years to create and develop and more importantly, be adopted for use.
• We therefore decided to create an extension on an existing scientific API and add compatibility for MHC environment rather than developing a completely new API.
• In addition, building off of an existing API means that there is already a user base developing applications that would then become suitable for MHC without modifying the application code.
• The **GNU Scientific Library (GSL)** was selected to form the basis of the MHC-API.
  – GSL is an extensive, free scientific library that uses a clean and straightforward API.
  – It provides support for the Basic Linear Algebra Subprograms (BLAS) which is widely used in applications today.
  – GSL is data-type compatible with the Vector, Signal, and Image Processing Library (VSIPL), which is becoming one of the standard libraries in the embedded world.
An Initial Micro-Heterogeneous Computing API

The Micro-Heterogeneous Computing API (MHC-API) provides support for the following areas of scientific computing:

- Vector Operations
- Matrix Operations
- Polynomial Solvers
- Permutations
- Combinations
- Sorting
- Linear Algebra
- Eigen Vectors and Eigen Values
- Fast Fourier Transforms
- Numerical Integration
- Statistics

Based on GNU Scientific Library (GSL)
Analytical Benchmarking & Code-Type Profiling in MHC

• As in HC, analytical benchmarking is still needed in MHC to determine the performance of each processing element for every MHC-API call the device supports relative to the host processor.
  – This information must be known before program execution begins to enable the scheduling algorithm to determine an efficient mapping of tasks.
  – Since all of the processing elements have different capabilities, scheduling would be impossible without knowing the specific capabilities of each element.

• While analytical benchmarking is still required, code-type profiling is not needed in MHC. (known from the API call used)
  – Since Micro-Heterogeneous computers use a dynamic scheduler that operates during run-time, there is no need to determine the types of processing an application globally contains during compile time.
  – This eliminates the need for special profiling tools and removes a step from the typical heterogeneous development cycle.

HC-7: Bill Scheidel’s MS Thesis
Formulation of Mapping Heuristics For MHC

- Such heuristics for MHC environment are dynamic; the mapping is done during runtime.
- The scheduler only has knowledge about those tasks that have already been scheduled.
- MHC environment consists of a set $Q$ of $q$ heterogeneous processing elements.
- $W$ is a computation cost matrix of size ($v \times q$) that contains the estimated execution time for all tasks already created where $v$ is the number of the task currently being scheduled.
- $w_{ij}$ is estimated execution time of task $i$ on processing element $p_j$.
- $B$ is a communication cost matrix of size ($q \times 2$), where $b_{i,1}$ is the communication time required to transfer this task to processing element $p_i$ and $b_{i,2}$ is the per transaction overhead for communicating with processing element $p_i$. The estimated time to completion (ETC) of task $i$ on processing element $p_j$ can be defined as:

$$ETC_{i,j} = w_{i,j} + b_{j,1} + b_{j,2}$$

- The estimated time to idle is the estimated amount of time before a processing element $p_j$ will become idle. The estimated time to idle (ETI) of processor $j$ is defined as:

$$ETI_j = \sum_{i=0}^{n} ETC_{i,j}$$
Initial Scheduling Heuristics Developed for MHC

• Three initial different schedulers are proposed for possible adoption and implementation in the MHC environment:
  • Fast Greedy
  • Real-Time (i.e. Dynamic) Min-Min (RTmm)
  • Weighted Real-Time Min-Min

• All of these algorithms are based on static heterogeneous scheduling algorithms which have been modified to fit the requirements of MHC as needed.

• These initial algorithms were selected based on the results of extensive static heterogeneous scheduling heuristics comparison found in published performance comparisons.
Fast Greedy Scheduling Heuristic

- The heuristic simply searches for the processor with the lowest $ETC$ for the task being scheduled. Tasks that have previously been scheduled are not taken into account at all in this algorithm.
- Fast Greedy first determines the subset of processors, $S$, of $Q$ that support the current task resulting from an MHC-API call being scheduled.
- The processor, $s_j$, with the minimum $ETC$ is chosen and compared against the $ETC$ of the host processor $s_0$.
- If the speedup gained is greater then $\gamma$ then the task is scheduled to $s_j$, otherwise the task is scheduled to the host processor.

Fast Greedy

1. Let $i$ represent the current task to be scheduled.
2. Determine that set of processors $S$ of size $s$ that support the function call $c_i$ where $c_i$ is the function that is to be executed. The host processor is denoted as $s_0$ and must always appear in $S$.
3. Calculate the $ETC$ of task $i$ for each of the $s$ processors.
4. Locate the processor $s_j$ where

   \[ ETC_{i,s_j} = \min_{0 \leq j \leq s} ETC_{i,j} \]

5. If $ETC_{i,s_j}/ETC_{i,s_0} > \gamma$ then schedule task $i$ on processor $s_j$ otherwise schedule task $i$ on processor $s_0$.
Dynamic Min-Min (RTmm) Scheduling Heuristic

- The RTmm first determines the subset of processing elements, \( S \), of \( Q \) that support the current task being scheduled.
- The \( ETC \) for the task and the \( ETI \) for each of the processing elements in \( S \) is then calculated.
- The \( ETC_{ij}(total) \) for task \( i \) on processing element \( p_j \) is equal to the sum of \( ETC_{ij} \) and \( ETI_j \). The processing element, \( s_j \), with the minimum newly calculated \( ETC_{total} \) is chosen and compared against the \( ETC_{total} \) of the host processor, \( s_0 \).
- If the speedup gained is greater than \( \gamma \) then the task is scheduled to \( s_j \), otherwise the task is scheduled to host processor, \( s_0 \).

---

Real-Time Min Min

1. Let \( i \) represent the current task to be scheduled.
2. Determine that set of processors \( S \) of size \( s \) that support the function call \( c_i \) where \( c_i \) is the function that is to be executed. The host processor is denoted as \( s_0 \) and must always appear in \( S \).
3. Compute the \( ETC \) of task \( i \) for each of the \( s \) processors.
4. Compute the \( ETI \) of processing element \( p_j \) for each of the \( s \) processors.
5. Compute \( ETC_{i,j}(total) = ETC_{i,j} + ETI \) for each of the \( s \) processors.
6. Locate the processor \( s_j \) where
   \[
   ETC_{i,s_j}(total) = \min_{0 \leq j \leq s} ETC_{i,j}(total)
   \]
7. If \( ETC_{i,s_j}(total)/ETC_{i,s_0}(total) > \gamma \) then schedule task \( i \) on processor \( s_j \), otherwise schedule task \( i \) on processor \( s_0 \).
Weighted Dynamic Min-Min (WRTmm)

- WRTmm uses the same algorithm as RTmm but adds two additional parameters so that the scheduling can be fine-tuned to specific applications. First, the parameter $\alpha$ takes into account the case when a task dependency exists for the task currently being scheduled. A value of $\alpha$ less than one tends to schedule these tasks onto the same processing element as the dependency, while a value greater than one tends to schedule these tasks onto a different processing element.

- Second, the parameter $\rho$ is used to schedule tasks to elements that support fewer MHC-API calls and must be between 0 and 1. A low value of $\rho$ informs the scheduler not to include the number of MHC-API calls supported by devices as a factor in the mapping decision, while the opposite is true for high values of $\rho$.

---

**Weighted Real-Time Min Min**

1. Let $i$ represent the current task to be scheduled.
2. Determine the set of processors $S$ of size $s$ that support the mHC function call $c_i$ where $c_i$ is the mHC function that is to be executed. The host processor is denoted as $s_0$ and must always appear in $S$.
3. Compute the $ETC$ of task $i$ for each of the $s$ processors.
4. Compute the $ETI$ of processing element $p_j$ for each of the $s$ processors.
5. Compute $ETC_{i,j(tot)} = ETC_{i,j} + ETI$ for each of the $s$ processors.
6. Compute $ETC_{weighted}$ for each of the $s$ processes which is defined as

$$ETC_{i,j(weighted)} = (ETC_{i,j(tot)} \times \alpha) \times [1 - (\rho \times \frac{1}{c})]$$

where $c$ is the total number of MHC function calls that $p_j$ supports.
7. Locate the processor $s_j$ where

$$ETC_{i,s_j(weighted)} = \min_{0 \leq j \leq s} ETC_{i,j(weighted)}$$

8. If $ETC_{i,s_j(weighted)}/ETC_{i,s_0(weighted)} > \gamma$ then schedule task $i$ on processor $s_j$ otherwise schedule task $i$ on processor $s_0$.
Modeling & Simulation of MHC Framework

- The aid in the process of evaluating design considerations of the proposed MHC architecture framework, flexible modeling and simulation tools have been developed.
- These tools allow running actual applications that utilize an initial subset of MHC-API on a simulated MHC node.
- The modeled MHC framework includes the user-code transparent mapping of MHC-API calls to MHC devices.
- Most MHC framework parameters are configurable to allow a wide range of design issues to be studied. The capabilities and characteristics of the developed MHC framework modeling tools are summarized as follows:
  - Dynamically linked library written in C and compiled for Linux kernel 2.4 that supports an initial subset of 60 MHC-API calls allows actual compiled C programs to utilize MHC-API calls and run on the modeled MHC framework.
  - Flexible configuration of the number and characteristics of the devices in the MHC environment, including the MHC-API calls supported by each device and device performance for each call supported.
  - Modeling of task device queues for MHC devices in the framework.
  - Flexible configuration of the buses used by MHC devices including the number of buses available and bus performance parameters including transaction overheads and per byte transfer time.
  - Flexibility in supporting a wide range of dynamic task/device matching and scheduling heuristics.
  - Simulated MHC device drivers that actually perform the computation required by the MHC-API call.
  - Modeling of MHC operating system call handlers that allow task creation as a result of MHC-API calls, task scheduling using dynamic heuristics, and task execution on the devices using the simulated device drivers.
  - Extensive logging of performance data to evaluate different configurations and scheduling heuristics.
  - A graphical user interface implemented in Qt to allow setting up and running simulations. It also automatically analyzes the log files generated by the modeled MHC framework and generates in-depth HTML reports.
Preliminary Simulation Results

- The MHC framework modeling and simulation tools were utilized to run actual applications using the modeled MHC environment to demonstrate the effectiveness of MHC. The tools were also used to evaluate the performance of the initial proposed scheduling heuristics, namely Fast Greedy, RTmm, WRTmm. Some of the applications written using MHC-API and selected for the initial simulations include:
  - A matrix application that performs some basic matrix operations on a set of fifty 100 x 100 matrices.
  - A linear equations solver application that solves fifty sets of linear equations each containing one hundred and seventy-five variables.
  - A random task graphs application that creates random task graphs consisting of 300 fine-grain tasks with varying depth and dependencies between them. This application was used to stress test the initial scheduling heuristics proposed and examine their fine-grain load-balancing capabilities.

- Different MHC device configurations were used including:
  - Uniform Speedup: A variable number of PCI-based MHC devices ranging from 0 to 6 each with 20x speedup
  - Different Speedup: A variable number of MHC devices from 0 to 6 with speedups of 20, 10, 5, 2, 1, and 0.5

HC-7: Bill Scheidel’s MS Thesis
Matrix Application Simulation Performance Results (uniform device speedup =20)

<table>
<thead>
<tr>
<th>Processing Elements (including host)</th>
<th>Speedup Over Host Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>3</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>1.4</td>
</tr>
<tr>
<td>5</td>
<td>2.8</td>
</tr>
<tr>
<td>6</td>
<td>2.8</td>
</tr>
<tr>
<td>7</td>
<td>2.8</td>
</tr>
</tbody>
</table>

HC-7: Bill Scheidel’s MS Thesis

EECC722 - Shaaban
Linear Equations Solver Simulation Performance Results (uniform device speedup =20)

- The simulation results indicate that Fast Greedy performs well for small number of devices (2-3), while WRTmm provides consistently better performance than the other heuristics examined.
- Similar conclusions were reached for the matrix application and devices with different speedups.
- This indicates that Fast Greedy is a possible choice when the number of MHC devices is small but WRTmm is superior when the number of devices is increased.

HC-7: Bill Scheidel’s MS Thesis
Random Task Graphs For Devices with Varying Device Speedups

• Test run to study the capability of the heuristics at load-balancing many fine-grain tasks.
• The results show that only WRTmm reduced the overall execution time as more devices were added.
• The results also show that WRTmm is able to take advantage of slower devices in the computation.

HC-7: Bill Scheidel’s MS Thesis
Random Task Graphs Performance For Four Devices with Varying Device Speedups

- This simulation was used to test the performance of the heuristics with slower devices.
- Both Fast Greedy and RTmm proved to be very dependent on the speedup of the devices in order to achieve good performance.
- WRTmm did not demonstrate this dependence at all and execution time changed very little as the device speedup was reduced.
- This is close to the ideal case since the task graphs were not composed of computationally intensive tasks whose execution time could be improved by any dramatic amount by increasing the device speedup alone.
- The poor load-balancing capabilities of Fast Greedy and RTmm actually are partially hidden as device speedups are increased.

![Graph showing execution times for different device speedups]
MHC Framework Implementation

Target Platform: Linux

• 32-bit x-86 Linux kernel version 2.4
• Hardware interfacing and inter-processes scheduling is best done in the kernel.
• The open source of the Linux kernel makes it ideal for adding the required MHC framework modifications.
• Freely available tools.
• De-facto standard OS of computer clusters.
MHC Framework Implementation

MHC System Layers Overview

Function Libraries

User Code

MCH_GSL  MCH_IMG

MHC API

MHC Kernel Module

Kernel

MHC Devices/Drivers

HC-8: Kim Schuttenberg’s MS Thesis, HC-9
MHC Framework Implementation

MHC Framework Components

• **User Space Components:**
  – MHC User Libraries
    • Function libraries
    • Device code libraries
  – MHC Base System Library
    • Dependency analyzer
    • Second level or batch scheduler
    • API for creating and managing tasks

• **Kernel Space Components:**
  – First level or online scheduler
  – Device task queues
  – MHC-compliant device drivers
  – Device status and statistics gathering

HC-8: Kim Schuttenberg’s MS Thesis, HC-9
#63 Lec # 12 Fall 2010 11-3-2010
MHC Framework
Implementation
MHC System
Model Overview

EECC722 - Shaaban
MHC Framework Implementation:

MHC User Libraries

- **Function Libraries:**
  - Define interfaces to the functions.
  - Contains general implementations
  - i.e: Image processing, GSL (gnu scientific library)

- **Device Libraries:**
  - List functions each device can execute
  - Contains execution time estimates
  - Defines commands and configurations used by the device drivers for each function
MHC Framework Implementation:

MHC Base System Library

- Provides user interface
- Tracks dependencies among the functions called
- Makes high level scheduling decisions
  - Second level or batch scheduler
MHC Kernel Module

- Manage Communication with device drivers
- Perform final scheduling decisions (first level scheduler)
- Enforce system policies on fairness between competing tasks
- Collect usage information and statistics to aid in scheduling.
Two Level Scheduling

• Some scheduling mechanisms have high resource utilization, and are unsuitable for use in the kernel. Thus better handled in user space

• However, because multiple processes may be accessing MHC concurrently, the final matching must occur in the kernel module.

• Fortunately, the last step of mapping is similar for most of the dynamic scheduling algorithms.

• Allows for a wide range of scheduling algorithms to be implemented.
MHC Framework Implementation:

Kernel Level Scheduler

First Level

- Limited access to information about the tasks context.
- Must fairly handle tasks from multiple competing processes
- Limited memory and computation time available.
- Also referred to as online-scheduler
MHC Framework Implementation:

Kernel Level Scheduler Implementation

- Maintains one FIFO queue for each MHC device.
- When it receives a task, uses a linear minimization equation to place it in the correct queue.
- Factors considered
  - Execution time of the task on each device
  - ETI - estimated time to idle of each device
  - Reconfiguration time required
  - A constant provided by the batch scheduler.
MHC Framework Implementation:

**Batch Scheduler**

*User Level or Second Level*

- Handles complexities invisible to the Online scheduler, such a dependency information.
- Can prioritize tasks.
- Alters parameters passed to online scheduler.
- Uses a standard function pointer interface, to allow users to easily implement their own scheduling heuristics.
- Implements more complicated algorithms.
- Can implement any algorithm, but cannot handle competing processes.
MHC Framework Implementation:

Dependency Analysis

- Needed in order to preserve correct output.
- Parallelly executed tasks must have the same result as if code is executed sequentially.
- Dependency analysis is made more difficult by the wide variety of inputs and outputs a task can take.
Dependency Analysis: Simplifying Requirements

- All parameters to tasks can be described as a sequential range of memory addresses in host processor system memory.
- Parameter type, size, and data direction (input, output, bidirectional) is encoded before a task is submitted.
Dependency Analysis: Memory Region Tracking

- Two algorithms implemented:
  - **Hash Table:**
    - Usable when memory regions do not overlap, and do not change sizes.
    - Very Fast
  - **Exclusive Binary Tree:**
    - When two regions overlap, the regions are broken into non-overlapping and completely overlapping sub-regions.
    - Each region tracks the order in which tasks can access it in order to preserve correct results.
    - The process of splitting regions is easily expressed as an insertion into a binary tree.
    - Search complexity can vary from $O(\log(N))$ for a search matching to only one region to $O(N)$ for a search matching many regions.
Control Flow in MHC
Task Execution

Primary User Thread

- User Function Call
  - Fill In Parameters Structure
  - Generate Device ETCs

- Allocate a task monitor
- Invoke Batch Scheduler, if requested
- Check for Data Dependencies*
- Build the DAG
- If Dependencies exist, Invoke a worker thread.

Library Worker Thread

1. Pass scheduling data to kernel
2. Choose a device
3. Enter task into device queue
4. Wait for execution
5. Transfer data/code to* device driver
6. Wait For Task Finish
7. Retrieve Results
8. Resolve Dependencies
9. Check for tasks freed by resolved dependencies
10. Invoke Worker Threads if necessary

Red Border Indicates Critical Section

MHC Framework Implementation

HC-8: Kim Schuttenberg’s MS Thesis, HC-9

EECC722 - Shaaban

2004

#75 Lec # 12 Fall 2010 11-3-2010
MHC Framework Timing Results

- Most Important Measurement: Scheduler Latency
  - Affects the task granularity
  - Ties up CPU.
Results: Perspective

PCI Connection latency 0.5us (Spec)
PCI DMA Latency: 168us (Experimental/Linux)
HyperTransport Latency: 0.035us (Spec)
HyperTransport DMA Latency 11.8us (Extrapolated)

- T = Execution Time on CPU
- L = Scheduler Latency + Connect Latency
- Per device speedup = 2

T/L = Ratio of execution time to overheads

MHC Framework Implementation

HC-8: Kim Schuttenberg’s MS Thesis, HC-9
EECC722 - Shaaban
Latency Results

Basic MHC Task Submission

MHC Framework Implementation

Latency Results

Average ~ 10usec

Task Submission Latency Histogram

HC-8: Kim Schuttenberg’s MS Thesis, HC-9
Latency: Task Submission with Online (Kernel) Scheduler

Latency: Task Submission with Dynamic Task Selection
Latency: Best Case Dependency analysis

Without memory region overlap: Hash table used

MHC Framework Implementation

HC-8: Kim Schuttenberg’s MS Thesis, HC-9

EECC722 - Shaaban
Latency: Worst Case Dependency analysis

With memory region overlap: Exclusive binary tree used
Conclusions

• The MHC framework developed is a viable platform for further study into MHC.

• The two-level scheduling implementation allows for a large number of scheduling algorithms to be explored.

• Based on measured scheduling latency, MHC can target a task grain size well under 100us for jobs with high complexity.

• With even a modest per-device speedup of 2, it is possible to see an overall performance gain with tasks that have a 20us GPP compute time.
Possible Future MHC Work

- Caching and Coherency (to utilize device private memory)
- Automatic Task Aggregation
- Memory Locking Option for Jobs with Small Memory Regions
- Support of Priority Queues
- ETC Prediction/Automatic Profiling
- Back Annotation of ETC Predictions to Improve Accuracy (i.e. modify/predict ETC dynamically)
- More/Better Batch Scheduling Heuristics
- "Stackable" Batch Schedulers
- Creation of MHC Compliant Hardware/Drivers/Libraries