MPC: Multi-Processor Computing Framework
Guest Lecture
Parallel Computing
CIS 410/510
Department of Computer and Information Science
MPC: Multi-Processor Computing Framework

Patrick Carribault, Julien Jaeger and Marc Pérache

JUNE 17-18, 2013 |
Context

- **Starting point: programming model used today**
  - Most used standards: MPI and/or OpenMP
  - Current architectures: petaflopic machines such as TERA100
  - Languages: C, C++ and Fortran
  - Large amount of existing codes and libraries

- **Main target: transition to new programming models for Exascale**
  - Provide efficient runtime to evaluate mix of programming models
  - Unique programming model for all codes and libraries may be a non-optimal approach
  - Provide smooth/incremental way to change large codes and associated libraries
  - Avoid full rewriting before any performance results
  - Keep existing libraries at full current performance coupled with application trying other programming model
  - Example: MPI application calling OpenMP-optimized schemes/libraries

- **Multi-Processor Computing (MPC)**
Multi-Processor Computing (MPC) framework

- Runtime system and software stack for HPC
- Project started in 2003 at CEA/DAM (PhD work)
- Team as of May 2013 (CEA/DAM and ECR Lab)
  - 3 research scientists, 1 postdoc fellows, 8 PhD students, 1 apprentice, 1 engineer
- Freely available at http://mpc.sourceforge.net (version 2.4.1)
  - Contact: marc.perache@cea.fr, patrick.carribault@cea.fr or julien.jaeger@cea.fr

Summary

- Unified parallel runtime for clusters of NUMA machines

Unification of several parallel programming models

- MPI, POSIX Thread, OpenMP, …

Integration with other HPC components

- Parallel memory allocator, patched GCC, patched GDB, HWLOC, …
RUNTIME OPTIMIZATION
• **Goals**
  - Smooth integration with multithreaded model
  - Low memory footprint
  - Deal with unbalanced workload

• **MPI 1.3**
  - Fully MPI 1.3 compliant

• **Thread-based MPI**
  - Process virtualization
  - Each MPI process is a thread

• **Thread-level feature**
  - From MPI2 standard
  - Handle up to MPI_THREAD_MULTIPLE level (max level)
  - Easier unification with PThread representation

• **Inter-process communications**
  - Shared memory within node
  - TCP, InfiniBand

• **Tested up to 80,000 cores with various HPC codes**
MPC Execution Model: Example #1 (MPI)

- Application with 1 MPI task
• Optimizations
  • Good integration with multithreaded models [EuroPar 08]
    - No spin locks: programming model fairness without any busy waiting
    - *Scheduler-integrated* polling method
    - *Collective communications* directly managed by the *scheduler*
  • Scheduler optimized for Intel Xeon Phi
  • Low memory footprint
    - Merge network buffer between MPI tasks [EuroPVM/MPI 09]
    - Dynamically adapt memory footprint (on going)
  • Deal with unbalanced workload: Collaborative polling (CP) [EuroMPI 12]
  • Experimental results: IMB (left-hand side) and EulerMHD 256 cores (right-hand side)
OpenMP

- **OpenMP 2.5**
  - OpenMP 2.5-compliant runtime integrated to MPC
  - Directive-lowering process done by patched GCC (C,C++,Fortran)
    - Generate calls to MPC ABI instead of GOMP (GCC OpenMP implementation)

- **Lightweight implementation**
  - Stack-less and context-less threads (*microthreads*) [HiPC 08]
  - Dedicated scheduler (*microVP*)
    - On-the-fly stack creation
  - Support of oversubscribed mode
    - Many more OpenMP threads than CPU cores

- **Hybrid optimizations**
  - *Unified* representation of *MPI tasks* and *OpenMP threads* [IWOMP 10]
  - Scheduler-integrated Multi-level polling methods
  - Message-buffer privatization
  - Parallel message reception
  - Large NUMA node optimization [IWOMP 12]
2 MPI tasks + OpenMP parallel region w/ 4 threads (on 2 cores)
PTreads

- **Thread library completely in user space**
  - Non-preemptive library
  - User-level threads on top of kernel threads (usually 1 per CPU core)
  - Automatic binding (kernel threads) + explicit migration (user threads)
  - MxN $O(1)$ scheduler
    - Ability to map M threads on N cores (with M>>N)
    - Low complexity

- **POSIX compatibility**
  - POSIX-thread compliant
  - Expose whole PThread API

- **Integration with other thread models:**
  - *Intel’s Thread Building Blocks (TBB)*
  - Small patches to remove busy waiting
  - Unified Parallel C (UPC)
  - Cilk
Memory Allocation on Linux System

Memory allocation

- Linux uses lazy allocation
- Small allocations (< 128kB) GLIBC uses buffers to avoid high frequency call to sbrk/brk
- Big allocations (>= 128kB) GLIBC uses mmap system calls
  Malloc calls are only virtual memory reservations
  The real memory allocations are performed during first touch

What appends during first touch:

- Hardware generates an interruption
- Jump to the OS
- Search of the related VMA and check reason of the fault
- Request a free page to NUMA free list
- Reset the page content
- Map the page in the VMA, update the page table
- Return to the process
- It was done for all 4K pages (262 144 times for 1GB)
Page fault scalability evaluation
Multithread based approach (1 thread per core)
Process based approach (1 process per core)

4*4 Nehalem-EP=128 cores (left hand side) and Xeon Phi (right hand side)
Memory Allocation Optimization Step 1: User Space

Goals
- Reduce the number of system calls
- Increase performance in multithreaded context with a large number of cores
- Maintain data locality (NUMA-aware)

Ideas
- Hierarchical memory allocator
- Increase memory buffer size
Memory Allocation Optimization Step 2: Kernel Space

**Diagnosis**
- 40% of a page fault time is due to zero-page

**Goals**
- Reuse page within a process \(\rightarrow\) avoid useless page cleaning
- Portable for large number of memory allocators (use the mmap semantics)

\[\text{mmap}(...\text{MAP\_ANON}...)\]  \[\text{mmap}(...\text{MAP\_ANON}|\text{MAP\_PAGE\_REUSE}...)\]

[Diagram showing memory management and processes]
## AMR Code Results on Dual-Westmere (2*6 cores)

### Kernel patch and standard 4K pages

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Kernel</th>
<th>Total (s)</th>
<th>Sys. (s)</th>
<th>Mem. (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glibc</td>
<td>Std.</td>
<td>143,89</td>
<td>8,53</td>
<td>3,3</td>
</tr>
<tr>
<td>MPC-NUMA</td>
<td>Std.</td>
<td><strong>135,14</strong></td>
<td><strong>1,79</strong></td>
<td><strong>4,3</strong></td>
</tr>
<tr>
<td>MPC-Lowmem</td>
<td>Std.</td>
<td>161,58</td>
<td>15,97</td>
<td>2,0</td>
</tr>
<tr>
<td>MPC-Lowmem</td>
<td>Patched</td>
<td>157,62</td>
<td>10,60</td>
<td>2,0</td>
</tr>
<tr>
<td>Jemalloc</td>
<td>Std.</td>
<td>143,05</td>
<td>14,53</td>
<td>1,9</td>
</tr>
<tr>
<td>Jemalloc</td>
<td>Patched</td>
<td>140,65</td>
<td>9,32</td>
<td>3,2</td>
</tr>
</tbody>
</table>

### Kernel patch and Transparent Huge Pages

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Kernel</th>
<th>Total (s)</th>
<th>Sys. (s)</th>
<th>Mem. (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glibc</td>
<td>Std.</td>
<td>149,77</td>
<td>12,92</td>
<td>4,5</td>
</tr>
<tr>
<td>MPC-NUMA</td>
<td>Std.</td>
<td><strong>137,89</strong></td>
<td><strong>1,86</strong></td>
<td><strong>6,2</strong></td>
</tr>
<tr>
<td>MPC-Lowmem</td>
<td>Std.</td>
<td>196,51</td>
<td>28,24</td>
<td>3,9</td>
</tr>
<tr>
<td>MPC-Lowmem</td>
<td>Patched</td>
<td>138,77</td>
<td>2,90</td>
<td>3,8</td>
</tr>
<tr>
<td>Jemalloc</td>
<td>Std.</td>
<td>144,72</td>
<td>14,66</td>
<td>2,5</td>
</tr>
<tr>
<td>Jemalloc</td>
<td>Patched</td>
<td>138,47</td>
<td>6,40</td>
<td>3,2</td>
</tr>
</tbody>
</table>
AMR Code Results on Dual-Westmere (2*6 cores)

Hera execution
12 mpi tasks, 1 core per task

Execution time (sec)

Native

VM-nopatch

VM-withpatch

Compute
System
Other
Conclusion on memory allocation

NUMA-aware thread-aware allocator

- User space:
  - Reduce the number of system calls
  - Keep data locality
  - Good performances with a large number of threads
  - Tradeoff between memory consumption and execution time
  - User space allocator in included within the MPC framework

- Kernel space:
  - Remove useless page cleaning
  - Portable
  - Increase performances for standard and huge pages
  - Useful within virtual machines

More details in:
PROGRAMMING MODELS
Extended TLS [IWOMP 11]

- Mixing of thread-based models require flexible data management
  - Design and implementation of Extended TLS (Thread-Local Storage)
- Cooperation between compiler and runtime system
  - Compiler part (GCC)
    - New middle-end pass to place variables to the right extended-TLS level
    - Modification of backend part for code generation (link to the runtime system)
  - Runtime part (MPC)
    - Integrated to user-level thread mechanism
    - Copy-on-write optimization
    - Modified context switch to update pointer to extended TLS variables
- Linker optimization (GLIBC)
  - Support all TLS modes
  - Allow Extended TLS usage without overhead
Extended TLS Application: Automatic Privatization

• Global variables
  ■ Expected behavior: duplicated for each MPI task
  ■ Issue with thread-based MPI: global variables shared by MPI tasks located on the same node

• Solution: Automatic privatization
  ■ Automatically convert any MPI code for thread-based MPI compliance
  ■ Duplicate each global variable

• Design & Implementation
  ■ Completely transparent to the user
  ■ New option to GCC C/C++/Fortran compiler (-fmpc_privatize)
  ■ When parsing or creating a new global variable: flag it as thread-local
  ■ Generate runtime calls to access such variables (extension of TLS mechanism)
  ■ Linker optimization for reduce overhead of global variable access

• On-going Intel compiler support for Xeon and MIC
HLS (Hierarchical Local Storage) [IPDPS 12]

- **Goal:** Allow to share data among MPI tasks
- **Require compiler support**
- **Allow to save memory (GBs) per node**
- **Example of one global variable named `var`**
- **Duplicated in standard MPI environment**
- **Shared with HLS directive**
  
  ```
  #pragma hls node(var)
  ```
- **Updated with HLS directive**
  
  ```
  #pragma hls single(var) { ... }
  ```

---

**Memory**

**Socket 0**

- L3
- L2
- L1
- Core 0

**Socket 1**

- L3
- L2
- L1
- Core 0

---

**Graph:**

- **MPC w/ HLS**
- **MPC**
- **OpenMPI**

**Graph Details:**

- **Number of cores:** 256, 512, 736
- **Per-node memory consumption (MB):**
  - 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000
  - 256, 512, 736
- **Goals**
  - Exploit CPU and accelerators within existing applications (minimize modifications)
  - Dynamic load balancing and good integration with other programming models

- **Strong link between scheduler and software cache**
  - Take into account computational node topology (NUMA and NUIOA)
  - Decrease host \( \rightarrow \) device memory transfers
  - Software cache for data reuse (dedicated eviction policy)
  - Multiple scheduling policies to keep data locality (efficient mixing of optimized library calls and user tasks)
Heterogeneous Scheduler

- **PN:** deterministic resolution of transport equation (2D w/ MPI)
- **Focus on most time-consuming function** (~90% of sequential execution time).

```
1 /* Part 1: Large matrix multiplications 1 */
2 GEMM [in: \( A_X(136 \times 136) \), \( B_X(1M \times 136) \)] [out: \( C_X(1M \times 136) \)]
3 GEMM [in: \( A_Z(136 \times 136) \), \( B_Z(1M \times 136) \)] [out: \( C_Z(1M \times 136) \)]

5 /* Part 2: Small matrix multiplications */
6 GEMM [in: \( A_X(136 \times 136) \), \( BL_X(1K \times 136) \)] [out: \( CL_X(1K \times 136) \)]
7 GEMM [in: \( A_Z(136 \times 136) \), \( BL_Z(1K \times 136) \)] [out: \( CL_Z(1K \times 136) \)]
8 GEMM [in: \( A_X(136 \times 136) \), \( BR_X(1K \times 136) \)] [out: \( CR_X(1K \times 136) \)]
9 GEMM [in: \( A_Z(136 \times 136) \), \( BR_Z(1K \times 136) \)] [out: \( CR_Z(1K \times 136) \)]
10 BARRIER

12 /* Part 3: Tasks */
13 TASK [in: \( D_X(1024) \), \( C_X(1M \times 136) \)] [out: \( E_X(1M \times 136) \)]
14 TASK [in: \( D_Z(1024) \), \( C_Z(1M \times 136) \)] [out: \( E_Z(1M \times 136) \)]

16 BARRIER
17 TASK [in: \( D_X(1024) \), \( CL_X(1K \times 136) \)] [in-out: \( E_X(1M \times 136) \)]
18 TASK [in: \( D_Z(1024) \), \( CL_Z(1K \times 136) \)] [in-out: \( E_Z(1M \times 136) \)]

19 BARRIER
20 TASK [in: \( D_X(1024) \), \( CR_X(1K \times 136) \)] [in-out: \( E_X(1M \times 136) \)]
21 TASK [in: \( D_Z(1024) \), \( CR_Z(1K \times 136) \)] [in-out: \( E_Z(1M \times 136) \)]
22 BARRIER

23 /* Part 4: Large matrix multiplications 2 */
24 GEMM [in: \( F_X(136 \times 136) \), \( E_X(1M \times 136) \)] [out: \( F_X(1M \times 136) \)]
25 GEMM [in: \( F_Z(136 \times 136) \), \( E_Z(1M \times 136) \)] [out: \( F_Z(1M \times 136) \)]
```

1. Large matrix-matrix multiply
2. Small matrix-matrix multiply
3. User-defined simple tasks
4. Large matrix-matrix multiply
Heterogeneous Scheduler

**PN: CPUs performance**
(\textit{double precision, 1536x1536 mesh, }N=15, \textit{36 iterations})

- **Sequential**: 974 seconds
- **Parallel (8 cores)**: 154.21 seconds

**Final speed-up CPUs vs. heterogeneous:** 2.65 x

**CPUs**: 8-core Intel Xeon E5620
**GPU**: 2 GPUs Nvidia Telsa GTX M2090

**PN: Heterogeneous performance**
(\textit{double precision, 1536x1536 mesh, }N=15, \textit{36 iterations})

- **Heterogeneous DGEMM**: 84.29 seconds
- **Heterogeneous DGEMM ans tasks**: 68.28 seconds
- **Same w/ multiple scheduling policies**: 58.33 seconds
- **Theoretical no-transfer performance**: 32.53 seconds

Tera-100 Hybrid Node
Emerging Programming Models

- Evaluation of current and emerging models

- Task-based model
  - Cilk
    - Cilk on the top of MPC
    - Evaluation of mix MPI + OpenMP + Cilk
  - OpenMP 3.X tasks
    - Prototype a task engine
    - How to mix multiple task models?

- PGAS
  - UPC
    - Berkeley UPC on the top of MPC

- Heterogeneous
  - OpenCL
    - Evaluation of language capabilities
  - OpenACC
    - Evaluation of an OpenACC implementation (compiler part in GCC with CUDA backend)
Debugging

• Goal: tools to help application and feature debugging

• Static analysis [EuroMPI 2013]
  - Extend GCC compiler to analyze parallel application (MPI, OpenMP and MPI +OpenMP)
  - Detect wrong usage of MPI (collective communications with control flow)

• Interactive debugging [MTAAP 10]
  - Provide a generic framework to debug user-level thread
    - Evaluated on MPC, Marcel, GNUPth
  - Provide a patched version of GDB
  - Collaboration with Allinea DDT
    - MPC support in Allinea DDT 3.0

• Trace-based dynamic analysis [PSTI 13]
  - Use traces to debug large-scale applications
  - Crash-tolerant trace engine
  - Parallel trace analyzer
MPI Collective-Communication Debugging

- Motivating examples

```c
void f ( int r ) {
    if( r == 0 )
        MPI_Barrier(MPI_COMM_WORLD);
    return;
}
```

```c
void h ( int r ) {
    if( r == 0 ) {
        MPI_Reduce(MPI_COMM_WORLD, ...);
        MPI_Barrier(MPI_COMM_WORLD);
    } else {
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Reduce(MPI_COMM_WORLD, ...);
    }
    return;
}
```

```c
void g ( int r ) {
    if( r == 0 )
        MPI_Barrier(MPI_COMM_WORLD);
    else
        MPI_Barrier(MPI_COMM_WORLD);
    return;
}
```

- Analysis: f and h are incorrect while g is correct
- Main idea: detect incorrect functions with a two-step method:
  1. Compile-time identification of conditionals that may cause possible deadlocks
  2. Runtime verification with code transformation
MPI Collective-Communication Debugging

Compile-time Verification

- Intra-procedural analysis (GCC plugin)
- Control flow graph (CFG)

```
void f ( int r ) {
  if( r == 0 )
    MPI_Barrier(MPI_COMM_WORLD);
  return;
}
```

Output
- Warnings (conditionals)
- Set O of collectives that may deadlock

Code transformation

- Insert a Check Collective function (CC)
  before each collective in O
- Insert CC before the return statement

```
void f ( int r ){
  MPI_Comm c; int n1, n2;
  if( r == 0 ) {
    MPI_Comm_split(MPI_COMM_WORLD,1,0,&c);
    MPI_Comm_size( c,&n1 );
    MPI_Comm_size( MPI_COMM_WORLD,&n2 );
    if ( n1 != n2 ) MPI_Abort();
    MPI_Comm_free(&c);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Comm_split(MPI_COMM_WORLD,0,0,&c);
  MPI_Comm_size( c,&n1 );
  MPI_Comm_size( MPI_COMM_WORLD,&n2 );
  if ( n1 != n2 ) MPI_Abort();
  MPI_Comm_free(&c);
  return;
}
```
Results on the NAS Parallel Benchmark (NASPB)

- Experiments on Tera-100
- Benchmark Description
  - Language: Fortran, C
  - Version 3.2, Class C

Static check results

Overhead of average compilation time with and without code instrumentation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># collective calls</th>
<th># warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>LU</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>SP</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>IS</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>CG</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>FT</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

What an user can read on stderr when compiling NASPB IS

is.c:1093:1: warning: STATIC-CHECK: MPI_Reduce may not be called by all processes in the communicator because of the conditional line 923
Check inserted before MPI_Reduce line 994
### Execution results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># collective calls</th>
<th>% instrumented collectives</th>
<th># calls to CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>9</td>
<td>78%</td>
<td>8</td>
</tr>
<tr>
<td>LU</td>
<td>14</td>
<td>14%</td>
<td>6</td>
</tr>
<tr>
<td>SP</td>
<td>8</td>
<td>75%</td>
<td>7</td>
</tr>
<tr>
<td>IS</td>
<td>5</td>
<td>40%</td>
<td>3</td>
</tr>
<tr>
<td>CG</td>
<td>2</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>FT</td>
<td>8</td>
<td>0%</td>
<td>0</td>
</tr>
</tbody>
</table>

**Execution-Time Overhead for NASPB (Strong Scaling)**
CONCLUSION/FUTURE WORK
Conclusion

• **Runtime optimization**
  - Provide widely spread standards
  - MPI 1.3, OpenMP 2.5, PThread
  - Available at [http://mpc.sourceforge.net](http://mpc.sourceforge.net) (version 2.4.1)
  - Optimized for manycore and NUMA architectures

• **Programming models**
  - Provide unified runtime for MPI + X applications
  - New mechanism to mix thread-based programming models: Extended TLS
  - MPI extension for data sharing: HLS
  - Evaluation of new programming models

• **Tools**
  - Debugger support
  - Profiling
  - Compiler support
Future Work

- **Stabilize/promote MPC framework**
  - Optimize manycore support
  - Support to users (CEA, IFPEN, DASSAULT, NNSA, …)
  - Distribute MPC (Sourceforge, Curie, TERA 100, …)

- **From petascale to exascale**
  - Language evaluation
    - PGAS
    - One sided
  - Hardware: optimization/evaluation
    - Intel Xeon Phi (MIC)
    - GPGPU
  - How to help application to go from petascale to exascale

- **Always propose debugging/profiling tools of new features**
References

2014
• J. Clet-Ortega, P. Carribault, M. Pérache, Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures, (To appear EuroPar’14)

2013
• J. Jaeger, P. Carribault, M. Pérache, Data-Management Directory for OpenMP 4.0 and OpenACC, (HeteroPar’13)
• S. Didelot, P. Carribault, M. Pérache, W. Jalby, Improving MPI Communication Overlap With Collaborative Polling, (Springer Computing Journal)
• E. Saillard, P. Carribault, D. Barthou. Combining Static and Dynamic Validation of MPI Collective Communications. (EuroMPI’13)
• J.-B. Besnard, M. Pérache, W. Jalby. Event Streaming for Online Performance Measurements Reduction, (PSTI’13)

2012
• S. Didelot, P. Carribault, M. Pérache, W. Jalby, Improving MPI Communication Overlap With Collaborative Polling, (EUROMPI’12)
• A. Maheo, S. Koliai, P. Carribault, M. Pérache, W. Jalby, Adaptive OpenMP for Large NUMA Nodes, (IWOMP’12)
2012
• M. Tchiboukdjian, P. Carribault, M. Pérache, *Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks*, (IPDPS’12)

2011
• P. Carribault, M. Pérache, H. Jourdren, *Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications* (IWOMP’11)

2010
• P. Carribault, M. Pérache, H. Jourdren, *Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC* (IWOMP’10)
• K. Pouget, M. Pérache, P. Carribault, H. Jourdren, *User Level DB: a Debugging API for User-Level Thread Libraries* (MTAAP’10)

2009
• M. Pérache, P. Carribault, H. Jourdren, *MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption* (EuroPVM/MPI'09)

2008
• F. Diakhaté, M. Pérache, H. Jourdren, R. Namyst, *Efficient shared-memory message passing for inter-VM communications* (VHPC'08)
• M. Pérache, H. Jourdren, R. Namyst, *MPC: A Unified Parallel Runtime for Clusters of NUMA Machines* (EuroPar'08)
• S. Zuckerman, M. Pérache, W. Jalby, *Fine tuning matrix multiplications on multicore*, (HiPC'08)