NUMA Support for Charm++

Christiane Pousa Ribeiro (INRIA)
Filippo Gioachin (UIUC)
Chao Mei (UIUC)
Jean-François Méhaut (INRIA)
Gengbin Zheng(UIUC)
Laxmikant Kalé (UIUC)
Outline

• Introduction
  – Motivation
  – NUMA Problem

• Support NUMA on Charm++

• First Results

• Conclusion and Future work
Motivation for NUMA Platforms

- The number of cores per processor is increasing
  - Hierarchical shared memory multiprocessors
  - cc-NUMA is coming back (NUMA factor)
  - AMD hypertransport and Intel QuickPath
NUMA Problem

- Remote access and Memory contention

- Optimizes:
  - Latency
  - Bandwidth

- Assure memory affinity
NUMA Problem

- Remote access and Memory contention
- Optimizes:
  - Latency
  - Bandwidth
- Assure memory affinity
NUMA Problem

- Remote access and Memory contention
- Optimizes:
  - Latency
  - Bandwidth
- Assure memory affinity
NUMA Problem

- Remote access and Memory contention
- Optimizes:
  - Latency
  - Bandwidth
- Assure memory affinity
NUMA Problem

- Remote access and Memory contention

- Optimizes:
  - Latency
  - Bandwidth

- Assure memory affinity
NUMA Problem

• Remote access and Memory contention

• Optimizes:
  – Latency
  – Bandwidth

• Assure memory affinity
NUMA Problem

- Memory access types:
  - Read and write
  - Different costs

- Write operations are more expensive
  - Special memory policies

- On NUMA, data distribution matters!
Memory Management on NUMA

- User data is stored on virtual pages

- Memory policy
  - Defines the binding between virtual and physical pages

- The chosen memory policy will impact on the performance of an application
Memory Affinity on Linux

• The actual support for NUMA on Linux:
  – Physical memory allocation:
    • First-touch: first memory access
  – NUMA API: developers do all!
    • System call to bind memory pages
    • Numactl, user-level tool to bind memory and to pin threads
    • Libnuma an interface to place memory pages on physical memory
Charm++ Parallel Programming System

- Portability over different platforms
  - Shared memory
  - Distributed memory

- Architecture abstraction => programmer productivity

- Virtualization and transparency
Charm++ Parallel Programming System

• High-level approach to make easier the development of parallel applications

http://charm.cs.uiuc.edu/research/charm/

• Applications are written in C++
  – some library calls are required
  – interface description language
Charm++ Parallel Programming System

• Data management:
  – Stack and Heap

• Memory allocation based on malloc

• Isomalloc: allows threads migration
  – based on mmap system call

• What about physical memory?
NUMA Support on Charm++

• Our approach
  – Memory policies to distribute data over the machine

• Based on three parts:
  – +maffinity option
  – NUMA-aware isomalloc
  – NUMA-aware memory allocator??
+maffinity option

• set memory policy for processes or threads

• Allows charm++ users to select:
  – NUMA nodes to distribute data
  – Memory policy to place data

• Transparent: no application source code modifications
+maffinity option

• Based on Linux NUMA system call
  – Bind, preferred and interleave are used in our implementation

• Must be used with +setcpuaffinity option
  – Assure that threads/processes are placed in the same set of nodes

• Integrated into charm++ runtime by Chao Mei and Gengbin Zheng
./charmrun prog +p6 +setcpuaffinity +coremap 0,2,4,8,12,13 +maffinity +memnodemap 0,0,1,2,3,3 +mempol preferred
./charmrun prog +p4 +setcpuaffinity +coremap 0,4,8,12 +maffinity

In this case memory affinity is automatically managed
First Results

• Presented on 2010 Charm++ Workshop

• Charm++ version:
  – 6.1.3
  – Next-step: compare with 6.2.0 version

• Applications:
  – Molecular2D
  – Kneighbor
First Results

• NUMA machine
• AMD Opteron
• 8 (2 cores) x 2.2GHz processors
• Cache L2 (2Mbytes)
• Main memory 32Gbytes
• Low latency for local memory access
• Numa factor: 1.2 – 1.5
• Linux 2.6.32.6
• NUMA machine
• Intel EM64T
• 4 (24 cores) x 2.66GHz processors
• Shared cache L3 (16MB)
• Main memory 192Gbytes
• Numa factor: 2.6
• Linux 2.6.27
NUMA-aware Isomalloc for AMPI

• Considers the memory subsystem of a NUMA machine to distribute data:
  – Reduces the number of remote accesses
  – Optimizes bandwidth usage

• When mapping a memory page to physical memory verify if it is a NUMA machine
NUMA-aware Isomalloc

• Application runtime information and architecture characteristics to dynamically decide data placement

• NUMA-aware Isomalloc mechanism has two steps:
  • retrieve information of the machine
  • place data considering the memory policy
NUMA-aware Isomalloc

- The first step is static and performed on charm++ initialization

- Isomalloc memory policies:
  - Node affinity
  - Node interleave
  - Node neighbors
Isomalloc memory policies

- Node Affinity: migrates data to assure memory affinity for threads

- Node interleave: uses all memory banks of the machine to place data

- Node neighbors: places data on the thread node neighbors
Performance Evaluation

• Evaluate the performance of the NUMA-aware Isomalloc:
  – Migration and Jacobi 3D benchmarks
  – BigDFT
  – Different NUMA machines
  – Cluster of NUMAs

• Charm++ version:
  – 6.2.0 – net-linux-x86_64
NUMA Platform

- Opteron
- AMD Opteron
- 8 (2 cores) x 2.2GHz processors
- Cache L2 (2Mbytes)
- Main memory 32Gbytes
- Low latency for local memory access
- Numa factor: 1.2 – 1.5
- Linux 2.6.32.6
NUMA Platform

- Intel Xeon
- Intel EM64T
- 4 (24 cores) x 2.66GHz processors
- Shared cache L3 (16MB)
- Main memory 192Gbytes
- Numa factor: 2.6
- Linux 2.6.27
Opteron Machine

Jacobi 3D Benchmark
20 iterations - 16 Vp

Number of PEs

Intel Machine

Jacobi 3D Benchmark
20 iterations - 96 Vp
Jacobi Benchmark

- When compared to the worst placement:
  - On Opteron Machine:
    - Up to 8% of improvement gains
    - More calls to MPI_migrate() similar results
    - More MPI_migrate() and vp more gains (up to 23%)
  
  - On Intel Machine:
    - Up to 20% of improvement gains
    - More MPI_migrate() and vp no gains
Opteron Machine

Jacobi 3D Benchmark
20 iterations - 16 Vp

Intel Machine

Jacobi 3D Benchmark
20 iterations - 96 Vp
Jacobi Benchmark

• When compared to the best placement:
  – On Opteron Machine:
    • Up to 9% of improvement gains
    • More calls to MPI_migrate() similar results
    • More MPI_migrate() and vp more gains (up to 15%)

  – On Intel Machine:
    • Up to 3% of improvement gains
    • More MPI_migrate() and vp no gains
BigDFT

• Massively parallel application:
  – Quantum mechanics
  – Computation of electronic structure
  – Ab-initio simulation

• European project:
  – Dr. Thierry Deutsch, CEA
  – Prof. Stefan Goedecker, Basel University
  – Prof. Xavier Gonze, Université Catholique de Louvain
  – Prof. Reinhold Schneider, Christian Albrecht Universität zu Kiel
BigDFT

- Fortran code relatively large (75,000 lines of code):
  - Main operations (70% of execution time):
    - BLAS and convolutions can be done on GPUs (CUDA and OpenCL)

- Benchmark of large platforms (PRACE)

- MPI task by core, each task can use GPUs

- Code very regular:
  - every task performs the same operation
### BigDFT - AMPI

#### Code source adaptation:
- Global and static/save variables in an multithread environment

<table>
<thead>
<tr>
<th>Global variables</th>
<th>static/save variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified by the runtime on the global variable table (.got)</td>
<td>They are not Identified by the runtime</td>
</tr>
<tr>
<td>Swapglobal manage automatically such variables</td>
<td>Modifications on BigDFT source code</td>
</tr>
</tbody>
</table>
BigDFT - AMPI

- All variables <<save>> should be modified:
  - Use a global variable that will be privatized by AMPI
  - <<save>> variables are placed in a Fortran module that will be privatized by swapglobal

- In this version GPUs are not considered
BigDFT – Load Balancing

- Source code modification to include MPI_migrate()

- One MPI_migrate() per iteration:
  - BigDFT has 4 iterations

- Performance evaluation of the migration:
  - impact in the application
  - NUMA-aware isomalloc memory policies
BigDFT - Results

• Charm++ version : 6.2.0

• Cluster of NUMAs :
  – 96 nodes – 768 cores
  – Intel Nehalem Processor – 2.93 GHz
  – Each node has two NUMA nodes

• charmrun +p512 BigDFT +isomalloc_sync +tcharm_stacksize 10000000 +vp512
BigDFT - Results

- Some improvements with NUMA-aware isomalloc for 512 vp
- Next step: performance evaluation on cluster of larger NUMA machines
Conclusions

- Charm++ performance on NUMA can be improved
  - +maffinity
  - NUMA-aware isomalloc for AMPI

- Performance evaluation studies on benchmarks and real application
Future Work

• Improve Isomalloc memory policies

• NUMA-aware memory allocator for chares
  – Filippo Gioachin (UIUC)
  – Chao Mei (UIUC)

• Optimize data locality by distributing them over the machine memory banks

• Some memory policies: applied considering the access mode (read, write or read/write)
Thank you