PTG: an abstraction for unhindered parallelism

Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra
Overview

• Why a New Abstraction?
• Data-Flow Programming
• Parameterized Task Graphs in PaRSEC
• Comparing PTG against Competing Abstractions
• Task Affinity and Scheduling in PaRSEC
• PaRSEC Performance
Why a New Abstraction?
Why a New Abstraction?
Why a New Abstraction?

- More processing units
Why a New Abstraction?

- More processing units
- Deeper memory hierarchy
Why a New Abstraction?

- More processing units
- Deeper memory hierarchy
- Memory distribution
Why a New Abstraction?

- More processing units
- Deeper memory hierarchy
- Memory distribution
- Heterogeneity:
Why a New Abstraction?

- More processing units
- Deeper memory hierarchy
- Memory distribution
- Heterogeneity:
  - Compute (GPU, FPGA, etc)
Why a New Abstraction?

- More processing units
- Deeper memory hierarchy
- Memory distribution
- Heterogeneity:
  - Compute (GPU, FPGA, etc)
  - Memory
Why not MPI + X?
Why not MPI + X?

- MPI + X: OpenMP, OpenACC, OpenCL, CUDA, etc
Why not MPI + X?

- MPI + X: OpenMP, OpenACC, OpenCL, CUDA, etc
- Deeply coupled:
Why not MPI + X?

- MPI + X: OpenMP, OpenACC, OpenCL, CUDA, etc
- Deeply coupled:
  - Data distribution
Why not MPI + X?

- MPI + X: OpenMP, OpenACC, OpenCL, CUDA, etc
- Deeply coupled:
  - Data distribution
  - Parallelism
Why not MPI + X?

- MPI + X: OpenMP, OpenACC, OpenCL, CUDA, etc
- Deeply coupled:
  - Data distribution
  - Parallelism
  - Load balancing
Coarse Grain Parallelism

- Coarse Grain Parallelism with explicit message passing
- Essentially serial code with some explicit calls to a communication library
- Communication/computation overlap hard to expose: must be specified explicitly by the programmer
- Tends to lead to bulk-synchronous parallel programs
Data-Flow Programming
Data-Flow Programming

- Work units modeled as a graph, rather than sequentially
- Edges define data flow
- Runtime can automatically schedule tasks and overlap communication/computation
Data-Flow Programming

- Units of work are tasks
- Programs are collections of tasks & data-flow
- Reduced control flow
Parameterized Task Graph (PTG)
Parameterized Task Graph

- Originally by Cosnard et al. (1995, 1999)
- Program as a collection of task classes
- Representation independent of problem size
PTG Task Classes

- Class name
- Parameters and valid value ranges
- Affinity (to data)
- Precedence constraints: data input/output & logic
- Code region
PTG Ping-Pong

PING(s)
  s = 0..max_steps-1
  : A(s)
  RW A0 <- A(s)
       -> A0 PONG(s)
  READ A1 <- (s != 0) ? PONG(s-1)
BODY verify_response(A0, A1); END

PONG(s)
  s = 0..max_steps-2
  : A(s+1)
  RW A0 <- A0 PING(s)
       -> A1 PING(s+1)
BODY /* do nothing on data */ END
PTG Ping-Pong

PING(s)

\[ s = 0..\text{max\_steps}-1 \]

: A(s)

RW \hspace{1em} A0 \leftarrow A(s)

\rightarrow A0 \hspace{1em} \text{PONG}(s)

READ \hspace{1em} A1 \leftarrow (s \neq 0) \ ? \ \text{PONG}(s-1)

BODY \hspace{1em} \text{verify\_response}(A0, A1); \hspace{1em} \text{END}

PONG(s)

\[ s = 0..\text{max\_steps}-2 \]

: A(s+1)

RW \hspace{1em} A0 \leftarrow A0 \hspace{1em} \text{PING}(s)

\rightarrow A1 \hspace{1em} \text{PING}(s+1)

BODY \hspace{1em} /* \text{do nothing on data} */ \hspace{1em} \text{END}
PTG Ping-Pong

PING(s)
  \[ s = 0..\text{max\_steps}-1 \]
  : A(s)
  RW A0 <- A(s)
  \[ \rightarrow A0 \text{ PONG}(s) \]
  READ A1 <- (s \neq 0) ? PONG(s-1)
BODY verify\_response(A0, A1); END

PONG(s)
  \[ s = 0..\text{max\_steps}-2 \]
  : A(s+1)
  RW A0 <- A0 PING(s)
  \[ \rightarrow A1 \text{ PING}(s+1) \]
BODY /* do nothing on data */ END
PTG Ping-Pong

PING(s)
  s = 0..max_steps-1
  : A(s)
  RW  A0 <- A(s)
      -> A0 PONG(s)
  READ A1 <- (s != 0) ? PONG(s-1)
BODY verify_response(A0, A1); END

PONG(s)
  s = 0..max_steps-2
  : A(s+1)
  RW  A0 <- A0 PING(s)
      -> A1 PING(s+1)
BODY /* do nothing on data */ END
PTG Ping-Pong

PING(s)
  s = 0..max_steps-1
  : A(s)
    RW   A0 <- A(s)
         -> A0 PONG(s)
    READ A1 <- (s != 0) ? PONG(s-1)
BODY verify_response(A0, A1); END

PONG(s)
  s = 0..max_steps-2
  : A(s+1)
    RW   A0 <- A0 PING(s)
         -> A1 PING(s+1)
BODY /* do nothing on data */ END
PTG Ping-Pong

PING(s)
  s = 0..max_steps−1
  : A(s)
  RW   A₀ <- A(s)
       -> A₀ PONG(s)
  READ A₁ <- (s != 0) ? PONG(s−1)
BODY verify_response(A₀, A₁); END

PONG(s)
  s = 0..max_steps−2
  : A(s+1)
  RW   A₀ <- A₀ PING(s)
       -> A₁ PING(s+1)
BODY /* do nothing on data */ END
PTG Ping-Pong

PING(s)
  s = 0..max_steps-1
  : A(s)
  RW   A0 <- A(s)
       -> A0 PONG(s)
  READ A1 <-- (s != 0) ? PONG(s-1)
BODY verify_response(A0, A1); END

PONG(s)
  s = 0..max_steps-2
  : A(s+1)
  RW   A0 <- A0 PING(s)
       -> A1 PING(s+1)
BODY /* do nothing on data */ END
PTG Comparisons
Dynamic Task Graph

• Asynchronous tasks generated by code at runtime

• Dynamic discovery of task graph

• Used by other task execution runtimes:
  • Legion
  • StarPU
  • OpenMP
  • PaRSEC, as an extension (see Hoque et al., ScalA17)
Dynamic Task Graph

for (k = 0; k < MT; k++) {
    Insert_Task( geqrt, A[k][k], INOUT, T[k][k], OUTPUT);
    for (m = k+1; m < MT; m++) {
        Insert_Task( tsqrt, A[k][k], INOUT | REGION_D | REGION_U,
                    A[m][k], INOUT | LOCALITY,
                    T[m][k], OUTPUT);
    }
    for (n = k+1; n < NT; n++) {
        Insert_Task( unmrq, A[k][k], INPUT | REGION_L,
                    T[k][k], INPUT,
                    A[k][m], INOUT);
        for (m = k+1; m < MT; m++) {
            Insert_Task( tsmqr, A[k][n], INOUT,
                        A[m][n], INOUT | LOCALITY,
                        A[m][k], INPUT,
                        T[m][k], INPUT);
        }
    }
}
DTG Drawbacks

- Task instances unknown prior to discovery
- Memory requirements grow with problem size; task instances require independent memory
- Skeleton program that submits tasks to runtime; must build DAG based on dynamic properties of the program
- Fixed-size window of executing tasks can be used to reduce memory requirements, but restricts parallelism
- Restricted by control flow adherence
PTG vs DTG: Chains

\[ c \times W \]

\[ W \]

Task
Data flow
Serial program
control flow
PTG vs DTG: Chains

for (i=0; i<W; i++) {
    Task1( RW:Data[i][0] );

    for (j=1; j<c*W; j++) {
        Task2( R:Data[i][j-1], W:A[i][j] );
    }
}

PTG vs DTG: Chains

Task1(i)
   i = 0..W-1
   : Data(i,0)
   A <- Data(i,0)
   -> A Task2(i,1)
BODY ... END

Task2(i,j)
   i = 0..W-1
   j = 1..c*W-1
   : Data(i,j)
   A <- (j == 1) ? A Task1(i)
     <- (j > 1) ? A Task2(i,j-1)
     -> (j < c*W-1) ? A Task2(i,j+1)
     -> Data(i,j)
BODY ... END
PTG vs DTG: Chains

\[ S_{DTG} = cW + (W - 1)(c - 1)W \]

\[ S_{PTG} = \frac{cW^2}{P} \]

\[ \text{Speedup} = \frac{S_{DTG}}{S_{PTG}} = P \left( 1 - \frac{1}{c} + \frac{1}{cW} \right) = O(P) \]
PTG vs CGP

- Doesn’t deal well (or at all) with varying parallelism
- Idle time: bulk synchronous and load imbalance / noise
- Communication/computation overlapping
- Memory-hierarchy-awareness loses portability
- Multiple models for compute heterogeneity: MPI + X
PTG vs CGP

QR factorization
Task Affinity and Scheduling
Task Affinity and Scheduling

- Task scheduling is a well-studied problem: NP-complete, efficient heuristics and approximations usually used.

- Tasks scheduled on nodes with task affinity hints.

- Within a node, several strategies are used:
  - Memory locality
  - Starvation minimization
  - User-defined priorities
Task Affinity and Scheduling

- Memory locality:
  - Hierarchy of ready task queues mapped to memory hierarchy: one per core/socket/node
  - Since child tasks are put into same queues as parent, this guarantees some level of memory locality
Task Affinity and Scheduling

- Starvation minimization:
  - Shared task queue ensures compute resources aren’t starved of tasks (and thus idle)
  - Antithetical to memory locality
Task Affinity and Scheduling

- Hybrid scheduling:
  - Short local queues improve locality of ready tasks
  - Excess ready tasks are placed on a shared queue, reducing starvation
  - User-provided priorities are versatile and can be used instead for regular algorithms that are well-understood
Performance
Performance

• Comparison with applications/libraries using MPI

• Several libraries:
  - LibSCI: vendor ScaLAPACK tuned for Cray
  - DPLASMA: dense linear algebra on top of PaRSEC
Performance

Solving Linear Least Square Problem (DGEQRF)

60-node, 480-core, 2.27GHz Intel Xeon Nehalem, IB 20G System
Theoretical Peak: 4358.4 GFlop/s

![Graph showing performance comparison between Hierarchical QR, DPLASMA DGEQRF, and LibSCI Scalapack.](image)
Performance

DGEQRF performance strong scaling
Cray XT5 (Kraken) - N = M = 41,472

![Graph showing performance scaling for DGEQRF with Cray XT5 (Kraken) showing comparison between PARSEC DPLASMA and LibSCI Scalapack.]
Performance

Distributed Hybrid DPOTRF Weak Scaling on Keeneland
1 to 64 nodes (16 cores, 3 M2090 GPUs, Infiniband 20G per node)

- PaRSEC DPOTRF (3 GPU per node)
- Ideal Scaling

Performance (TFlop/s)

Number of Cores+GPUs

N=31k

N=246k
Performance

Execution Time of icsd_t2_8() subroutine in CCSD of NWChem

- Original (isolated) icsd_t2_8()
- PaRSEC (isolated) icsd_t2_8()

 Nodes x Cores/Node

<table>
<thead>
<tr>
<th></th>
<th>16x1</th>
<th>16x2</th>
<th>16x4</th>
<th>16x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.0</td>
<td>5.0</td>
<td>4.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

- Current and upcoming HPC systems will require a new abstraction to take full advantage of.
- PTG is proposed as the solution:
  - More flexible
  - Exposes more parallelism
  - Lower overheads than DTG
Questions?