Compilation Techniques and Language Extensions for Demand-Driven Execution on Heterogeneous Architectures

Introduction

Imperative programming languages are still the most popular programming language paradigm. Hybrid programming models in HPC: MPI + OpenMP + CUDA isn’t going away.

Extracting all available parallelism from real applications is extremely difficult:
- Real-world applications that execute on massively parallel architectures are complex and difficult for the compiler to analyze (e.g. irregular data access patterns).

Asynchronous programming models show real promise:
- Intel CNC, Plasma, ETI Swarm, etc.
- Data-parallel combined with demand-driven programming paradigms can help with this.

How do we support today’s programmers in leveraging massively parallel architectures?
- Incrementally move from imperative models to demand-driven
- “Legacy” applications do not need to be rewritten from scratch.

Goals

Develop compilation techniques to efficiently target demand-driven execution models from Chapel:
- Examples of demand-driven execution models include ETI Swarm, StarPU, etc..

Should be applicable to any Imperative programming language with parallel facilities:
- The Chapel programming language is used as a proof of concept and not strictly necessary.

Introduce new language extensions to generalize all forms of parallel constructs:
- Parallel constructs include: data-, task-, nested-, and pipeline-based parallelism
- User expresses ordering constraints between statements, loops, blocks of code, etc.
- Compiler constructs and outputs a task-graph that is “fed” into one of the demand-driven execution models.
- Leverages our previous work of generating GPU/Multicore-specific code from Chapel.

Compilation Techniques

Codelet Partitioning for Parallel Loops
Partition an imperative program written with explicitly parallel constructs into a task-graph.

Multi-Nested Parallel Loop Support
Perform partitioning techniques on programs with multi-nested parallel loops into a task-graph form.

Task-Parallel Construct Support
Support for OpenMP, CUDA, Intel CNC, Plasma, ETI Swarm, etc. by edges connecting nodes together.

High Level Approach

1. Insert dummy edges to prevent nodes from collapsing
2. Perform interval analysis
3. Remove dummy edges
4. Generate code

Compiler Optimizations

Task-Graph Node Coarsening (Fusion)
- Similar to static loop chunking as in OpenMP.
- task-graph node instances can be chunked

Dependency-based transformations
- If forall loops are nested inside of a sequential loop, task creation overhead is high.
- Perform loop interchange so that the parallel loop is in outermost level.
- Requires dependence analysis.

Language Extensions to Support Generalized Parallel Constructs

Programmer writes statements with additional annotations.

Annotations express ordering constraints:
- Compiler generates corresponding task-graph.

Control dependences are computed implicitly using techniques described earlier.

Example: Tiled Cholesky Factorization

Task-Graph Node Coarsening (Fusion)

Compiler

Task

Node

1. Chapel Source
2. Task-Graph
3. Generated Parallel Intermediate Language

High Level Approach

1. Perform interval analysis to isolate explicitly parallel loops from each other
2. Convert each interval into its associated task nodes. Control dependence is specified by edges connecting nodes together
3. Generate parallel intermediate language (PIL) code to map task-graph onto demand-driven execution model. This code will then be able to execute on both the multicore and GPU resources that are available.

Conclusions

- Developed compilation techniques to partition parallel imperative programming languages to demand-driven execution models.
- Introduced new language annotations to specify ordering constraints between sections of code.
- Facilitate the generation of task-graphs that are then mapped onto heterogeneous architectures.

Acknowledgments

This material is based upon work supported by National Science Foundation under Awards CNS 1111147, CCF 070226, and CSR-AED 072093: DARPA under UHPC Contract Number HR0011-10-3-0007; a grant from Cray; and by a 2011-2012 NVIDIA Research Fellowship Award.