An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism

AllScale

Enable developers to be productive and to port their applications to any scale of system

Thomas Fahringer, Herbert Jordan, Peter Thoman
University of Innsbruck, Austria
In Summary

- What people think of HPC programming
- How computer scientists deal with the algorithm
- How domain scientists construct their algorithm
- Our future of HPC Programming
Parallel Architectures

Multicore:

- OpenMP/Cilk

Accelerators:

- OpenCL/CUDA

Clusters:

- MPI/PGAS
Real World Architectures

OpenMP/Cilk

OpenCL/CUDA ≠

MPI/PGAS
Hybrid Codes

- *e.g.* MPI+X+Y

**Issues:**
- hard-coded problem *decomposition*
- lack of coordination among runtime systems

**Limited built-in support for:**
- portability, auto-tuning, load balancing, monitoring, or resilience
AllScale Vision

Application → Parallel Algorithm

Unified Parallel Programming Model

Toolchain → Portability, Tuning, and Resilience

Multicore → Accelerators → Clusters → Heterogeneous Clusters

Memory

● C

● C

● G

● M

● $
Conventional Flat Parallelism

How to map flat parallelism to a hierarchical parallel architecture?
Complex handling of errors – global operations

A_t=N

A_t=0

parallelism

time

linear parallel growth

... global barrier
AllScale Core Programming Model

• Try to provide an automatic solution:
  – Performance portability, load balancing, resilience, autotuning

• Our answer:
  Recursive Nested Task Parallelism
  – Why?
Recursively Nested Parallelism

Recursive call

Exponential parallel growth

Global Synchronisation
Local Synchronisation

... Recursive call
Objective

• **Developers:**
  – focus on application
  – expose maximum amount of parallelism

• **Toolchain:**
  – utilize parallelism
  – handle data management and portability
  – load balancing, resilience, and tuning
API

• Based on C++ templates
  – Widely used industry standard

• Two Layers:

  **User-Level API**
  • High-level abstractions (e.g. grids, meshes, stencils, channels)
  • Familiar interfaces (e.g. parallel for loops, map-reduce)

  **Core API**
  • Generic function template for recursive parallelism
  • Set of recursive data structure templates
  • Synchronization, control- and data-flow primitives
auto allscale_fib = allscale::prec(
    [](int n) { return n<2; },
    [](int n) { return n; },
    [](int n, const auto& fib) {
        auto x = fib(n-1);
        auto y = fib(n-2);
        return x.get() + y.get();
    });
No Recursion Required

• Previous code directly uses core API and is one of the smallest possible examples
• You probably have (at least) two questions:
  – *What about data?*
  – *How am I supposed to write a recursively task parallel version of my HPC code?*
What about data?

• The AllScale environment manages data for you
  – Whether to distribute it, keep it up to date, move it to an accelerator, make a backup for resilience, ...

• What it needs for that is a data item type $T$, which specifies the following types:
  – a type $F$ for fragments of the data storage
  – a type $R$ for addressing sub-ranges of the data structure

Domain scientists are not expected to write these! They are part of the user API.
How to write a recursively task parallel version for an HPC code?

- The short answer: you don’t need to.
- There are three options:
  - Directly use allscale::prec.
  - Use mid-level primitives provided by the user API. (e.g. allscale::pfor)
  - Use high-level algorithmic skeletons which fit your application domain (also part of the user API).
The diagram illustrates the `pfor` operator, which is a generic function used for parallel loops in C++. It takes three arguments: the start of the iterator range (inclusive), the end of the iterator range (exclusive), and the body function (in this case, a C++11 lambda). The lambda function captures the array `A` by reference and initializes the first 10 elements of `A` with values 0-9 in parallel.

The code example shown in the diagram is:

```cpp
pfor(0,10,[&](int i) {
    A[i] = i;
});
```

This initializes the first 10 elements of array `A` with values 0-9 in parallel.
pfor Implementation

- pfor construct
  - <uses>
  - open AllScale User API

- prec operator & treetures
  - <provides>
  - closed AllScale Core API

- Compiler & Runtime System
  - Toolchain implementation

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 671603
Interfaces

Application

Generic Parallel Primitives (C++ Template API)

User-Level API
Core API

API-aware high-level Compiler

Unified Runtime System
Scheduler

Online Monitoring and Analysis

Resilience Management

Standard C++ Toolchain

Desktop Hardware

Small- to Extreme-Scale Parallel Architectures

Development

Tuning & Deployment

Pilot Applications

Single Source User Interface

Generic APIs for abstract Algorithm Descriptions

Code Generation for Accelerators and Distributed Memory

Universal Abstract Machine Model

Dynamic Load, Data and Resource Management

Parallel Hardware

Computation & Data Management

Decomposition & Restructuring

Identify & Express Parallelism
Execution

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 671603
AllScale Products

Parallel C++
Data Structures
and Algorithms

Compiler and
Runtime System
providing
Portability, Tuning, and Resilience
Objective

• **Developers:**
  – focus on application
  – expose maximum of parallelism

• **Toolchain:**
  – utilize parallelism
  – handle data management and portability
  – load balancing, resilience, and tuning
### Pilot Applications

<table>
<thead>
<tr>
<th>iPIC3D</th>
<th>AmDaDos</th>
<th>Fine/Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit particle in-cell code for space weather applications</td>
<td>Adaptive meshing, data assimilation for dispersion of oils spills</td>
<td>Large Industrial unsteady CFD simulations</td>
</tr>
<tr>
<td>KTH</td>
<td>IBM Research</td>
<td>NUMECA</td>
</tr>
</tbody>
</table>

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 671603
Summary

• **Challenge**
  – Explore recursive task parallelism for extreme scale HPC

• **AllScale**
  – single programming model based on C++ templates
  – main source of parallelism: recursive parallelism
  – single compiler/single runtime system
  – auto-tuning, code-versioning, fault tolerance, on-line monitoring

• **First prototype released with tutorial**
  [https://github.com/allscale](https://github.com/allscale)

• **More information**
  – [www.allscale.eu](http://www.allscale.eu)
Partners

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 671603