PaRSEC: Distributed task-based runtime for scalable hybrid applications

George Bosilca and many others

Intertwine Workshop, April 20, 2018

https://bitbucket.org/icldistcomp/parsec
**PaRSEC: a generic runtime system for asynchronous, architecture aware scheduling of fine-grained tasks on distributed many-core heterogeneous architectures**

**Concepts**
- Clear separation of concerns: compiler optimize each task class, developer describe dependencies between tasks, the runtime orchestrate the dynamic execution
- Interface with the application developers through specialized domain specific languages (PTG/JDF/TTG, Python, insert_task, fork/join, ...)
- Separate algorithms from data distribution
- Make control flow executions a relic

**Runtime**
- Portability layer for heterogeneous architectures
- Scheduling policies adapt every execution to the hardware & ongoing system status
- Data movements between producers and consumers are inferred from dependencies. Communications/computations overlap naturally unfold
- Coherency protocols minimize data movements
- Memory hierarchies (including NVRAM and disk) integral part of the scheduling decisions
PaRSEC = a data centric programming environment based on asynchronous tasks executing on a heterogeneous distributed environment

- An execution unit taking a set of input data and generating, upon completion, a different set of output data.
- Tasks and data have a coherent distributed scope (managed by the runtime)
- Low-level API allowing the design of Domain Specific Languages (JDF, DTD, TTG)
- Supports distributed heterogeneous (and trigger tasks execution) environments.
  - Communications are implicit (the runtime moves data)
  - Resources are dynamic (threads, accelerators)
- Built-in resilience, performance instrumentation and analysis (R, python)
Software design based on Modular Component Architecture (MCA) of Open MPI.

- Clear components API
- Runtime selection of components
- Implementing a new component has little impact on the rest of the software stack.
A data is a manipulation token, the basic logical element (view) used in the description of the dataflow
- Locations: have multiple coherent copies (remote node, device, checkpoint)
- Shape: can have different memory layout
- Visibility: only accessible via the most current version of the data
- State: can be migrated / logged

**Data collections** are ensemble of data distributed among the nodes
- Can be regular (multi-dimensional matrices)
- Or irregular (sparse data, graphs)
- Can be regularly distributed (cyclic-k) or user-defined

**Data View** a subset of the data collection used in a particular algorithm (aka. submatrix, row, column,...)

A data-copy is the practical unit of data
- Has a memory layout (think MPI datatype)
- Has a property of locality (device, NUMA domain, node)
- Has a version associated with
- Multiple instances can coexist
DSL: The PaRSEC application

Define a distributed collection of data (here 1 dimension array of integers)

Start PaRSEC (resource allocation)

Create a tasks placeholder and associate it with the PaRSEC context

Add tasks. A configurable window will limit the number of pending tasks

Wait ’till completion

parsec_context_t* parsec;
parsec = parsec_context_init(NULL, NULL); /* start the PaRSEC engine */

parsec_handle_t* parsec_dtd_handle = parsec_handle_new (parsec);
parsec_enqueue(parcsec, (parsec_handle_t*) parsec_dtd_handle);
parsec_context_start(parcsec);

parsec_vector_t dDATA;
parsec_vector_init( &dDATA, matrix_Integer, matrix_Tile,
                   nodes, rank,
                   1, /* tile_size*/
                   N, /* Global vector size*/
                   0, /* starting point */
                   1 ); /* block size */

parsec_dtd_handle 0 /* DONE */);
How to describe a graph of tasks?

- Uncountable ways
  - Generic: Dagguer (Charm++), Legion, ParalleX, Parameterized Task Graph (PaRSEC), Dynamic Task Discovery (StarPU, StarSS), Yvette (XML), Fork/Join (spawn). CnC, Uintah, DARMA, Kokkos, RAJA
  - Application specific: MADNESS, ...

- PaRSEC runtime
  - The runtime is agnostic to the domain specific language (DSL)
  - Different DSL interoperate through the data collections
  - The DSL share
    - Distributed schedulers
    - Communication engine
    - Hardware resources
    - Data management (coherence, versioning, ...)
  - They don’t share
    - The task structure
  - The internal dataflow depiction
DSL: The insert_task interface

Define a distributed collection of data (here 1 dimension array of integers)

Start PaRSEC (resource allocation)

Create a tasks placeholder and associate it with the PaRSEC context

Add tasks. A configurable window will limit the number of pending tasks

Wait 'till completion

```c
DSL: The insert_task interface

Define a distributed collection of data
(here 1 dimension array of integers)

Start PaRSEC (resource allocation)

Create a tasks placeholder and associate it with the PaRSEC context

Add tasks. A configurable window will limit the number of pending tasks

Wait 'till completion

```
The Parameterized Task Graph (JDF)

- A dataflow parameterized and concise language
- Accept non-dense iterators
- Allow inlined C/C++ code to augment the language [any expression]
- Termination mechanism part of the runtime (i.e. needs to know the number of tasks per node)
- The dependencies had to be globally (and statically) defined prior to the execution
- Dynamic DAGs non-natural
- No data dependent DAGs
Experiments on Arc machines,
• E5-2650 v3 @ 2.30GHz
• 20 cores
• gcc 6.3
• MKL 2016
• PaRSEC-2.0-rc1
• StarPU 1.2.1
• CUDA 7.0

MAGMA runs out of GPU memory

B: big tile size
b: small tile size
ib: inner block size
Dense Linear Algebra

DPLASMA = ScaLAPACK + runtime (PaRSEC)

h stands for dynamic Hierarchical algorithms (a task can divide itself)
Overhead of insert_task

Cholesky (double precision) on Haswell, 8 nodes, 20 cores/node, 160 cores

PaRSEC-PTG, Tile size: 128
PaRSEC-DTD, Tile size: 128
PaRSEC-DTD, 128, Execution Only

PaRSEC-PTG, Tile size: 320
PaRSEC-DTD, Tile size: 320
PaRSEC-DTD, 128, Execution Only
Overhead of insert_task

\[ T_{PTG} = \frac{N \times C_T}{P \times n} \]

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + \frac{N \times C_D}{P} + \frac{N \times C_R}{P} \]

- \( T_{DTD/PTG} \): Overall time
- \( N \): Total number of tasks
- \( C_T \): Cost/duration of each task
- \( P \): Total number of nodes/process
- \( n \): Total number of cores
- \( C_D \): Cost of discovering a task
- \( C_R \): Cost of building DAG/relationship

Diagram: Cholesky (double precision) on Haswell, 8 nodes, 20 cores/node, 160 cores

PaRSEC-PTG, Tile size: 128
PaRSEC-DTD, Tile size: 128
PaRSEC-DTD, 128, Execution Only

PaRSEC-PTG, Tile size: 320
PaRSEC-DTD, Tile size: 320
PaRSEC-DTD, 128, Execution Only

Graph shows Gflops (Gigaflops) vs. Size(N)/No. of Tasks (millions) for different tile sizes and configurations.
**Overhead of insert\_task**

\[ T_{PTG} = \frac{N \times C_T}{P \times n} \]

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + \frac{N \times C_D}{P} + \frac{N \times C_R}{P} \]

Benefits: critical path is defined by the sequential ordering

Drawbacks: impossible to build collective patterns, selecting the window size is difficult, all data movement must be known globally (and their order is critically important)

- There are three types of scenario
  - Insert All: Each rank inserts all tasks, and executes only locals
  - Select Insert: Each rank inserts only local tasks, but iterates over all tasks.
  - Insert Local: Each rank only inserts local tasks.
Task dependencies done on the coordination doorbells instead of data
Need for lookahead to compensate for the high level lack of parallelism

\[ A(k, k) \rightarrow A(k+1 .. N, k), \ldots \]
**Natural data-dependent DAG Composition**

Example \( \text{POTRI} = \text{POTRF} + \text{TRTRI} + \text{LAUUM} \)

- **POTRF**
- **TRTRI**
- **LAUUM**

### 3 approaches:

- **Fork/Join**: complete POTRF before starting TRTRI
- **Compiler-based**: give the three sequential algorithms to the Q2J compiler, and get a single PTG for POINV
- **Runtime-based**: tell the runtime that after POTRF is done on a tile, TRTRI can start, and let the runtime compose

Traditional

PaRSEC
More dynamic applications

Dynamically redistribute the data
- use PAPI counters to estimate the imbalance
- reschedule the frontiers to balance the workload

Interoperability between GlobalArray + PaRSEC + MPI

Figure: Elastic wave propagation in 3D (2D slice view)
Dynamic distributed Workload

- **Active process**: process has pending actions or has pending emissions to perform
- **Idle process**: no further pending local actions
  - Idle process returns to active state upon reception of a message
- **Workload termination** is a global state when in a snapshot all processes are idle and there are no in-transit messages

Nodes in the tree are tasks, and they are distributed across P processes

- **Wave Algorithms (4C)**
  - Once idle a process start a termination wave counting communications (sent and recv). Two identical waves are necessary to detect termination.

- **Optimal Delay Algorithms (EDOD)**
  - Reduce the idleness to the root of a tree (a non-leaf become idle when all children are idle + local). Acknowledge message reception to allow processes to become idle (follow the same tree and inverse the parents idleness)

- **Credit Distribution Algorithms (CDA)**
  - Spread credits via messages. Return credits to an entity once idle. Detect termination once all the credits are accounted for.

We have a forest of trees, each tree representing a different function or intermediary expression to compute

\[ \int_0^1 (g(x) + f(x)) \times g(x) - g(f(x)) \, dx \]
Random Walk application

\[ E(4C) \geq \frac{1}{q} \frac{2P}{\log(P)} + P + o(P) \]
\[ E(EDOD) \geq \frac{1}{q} \times 3 \log(P) + P + o(P) \]
\[ E(CDA) \leq 2P \]

Implementation in PaRSEC (exp. on Mira)

- MADNESS style trees
  - Threshold define the level of refinement (accuracy of the operations)
  - 10^8 nodes in the tree (aka tasks)
- CDA \( \frac{1}{2} \) the control messages compared with 4C
  - All of them are FLUSH (no BORROW)
Conclusions

• Programming can be made easy(ier)
  • Portability: inherently take advantage of all hardware capabilities
  • Efficiency: deliver the best performance on several families of algorithms
  • Domain Specific Languages to facilitate development
  • Interoperability: data is the centric piece

• Build a scientific enabler allowing different communities to focus on different problems
  • Application developers on their algorithms
  • Language specialists on Domain Specific Languages
  • System developers on system issues
  • Compilers on optimizing the task code

• Interact with hardware designers to improve support for runtime needs
  • HiHAT: A New Way Forward
    for Hierarchical Heterogeneous Asynchronous Tasking
Priority based Wish List

• Portable and efficient API/library for accelerator support
• Portable, efficient and interoperable communication library (UCX, libFabric, ...)
• Moving away from MPI will require an efficient datatype engine
  • Also supported by rest of the software stack (for interoperability)
• Resource management/allocation system
  • PaRSEC supports dynamic resource provisioning, but we need a portable system to bridge the gap between different programming runtimes
• Memory allocator: thread safe, runtime defined properties, arenas (with and without sbrk). (memkind?)
• Generic profiling system, tools integration
• Any type of task-based debugger and performance analysis