DPS – Dynamic Parallel Schedules

Sebastian Gerlach, R.D. Hersch
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

http://dps.epfl.ch
Parallel 3D tomographic slice server

http://visiblehuman.epfl.ch

Client PC

Internet

Server

IEEE Concurrency, April–June 99, 28–37
Parallel 3D tomographic slice server

Slice specification

Extraction of digital slice parts

Extracted slice parts

Resampled slice parts merged into the final displayable slice
Parallel 3D tomographic slice server

- **Client thread**: Compute the extents intersecting the slice.
- **Disk thread**: Read extent from disk.
- **Compute thread**: Extract slice part from extent.
- **Client thread**: Merge slice parts into a full slice. Visualize full slice.
Context

Cluster server systems: applications dynamically release or acquire resources at run time
Challenges

Execution graph made of compositional Split–Compute–Merge constructs

Split into subtasks

Compute

Merge results

Task farm

Generalization
Challenges

Creation of compositional Split–Operation–Merge constructs at run time
Challenges

Reusable parallel components at run time

Parallel service component

Parallel application calling the parallel service component
Overview

- DPS design and concepts
- Implementation
- Application examples
What is DPS?

- High-level abstraction for specifying parallel execution behaviour
- Based on generalized split-merge constructs
- Expressed as a directed acyclic graph of sequential operations
- Data passing along the graph are custom Data Objects
The graph describing application flow is called flowgraph.

Illustrates three of the basic DPS operations:

- Split Operation
- Leaf Operation
- Merge Operation

Implicit pipelining of operations
Split/Merge Operation Details

- The **Split** operation takes one data object as input, and generates multiple data objects as output.
- Each output data object represents a subtask to execute.
- The results of the tasks are collected later in a corresponding **Merge** operation.
- Both constructs are can contain user-provided code, allowing for high flexibility in their behavior.
Mapping concepts

- The flowgraph only describes the sequencing of an application.

- To enable parallelism, the elements of the graph must be mapped onto the processing nodes.
  - Operations are mapped onto Threads, grouped in Thread Collections.

- Selection of Thread in Thread collection is performed using routing functions.
The Thread Collections are created and mapped at runtime by the application.
Routing

- Routing functions are attached to operations
- Routing functions take as input
  - The Data Object to be routed
  - The target Thread Collection
- Routing function is user-specified
  - Simple: Round robin, etc.
  - Data dependent: To node where specific data for computation is available
Threads

• DPS Threads are also data structures
  – They can be used to store local data for computations on distributed data structures
  – Example: n-Body, matrix computations

• DPS Threads are mapped onto threads provided by the underlying Operating System (e.g. one DPS Thread = one OS Thread)
Flow control and load balancing

• DPS can provide flow control between split-merge pairs
  – Limits the number of data objects along a given part of the graph

• DPS also provides routing functions for automatic load balancing in stateless applications
Runtime

• All constructs (schedules, mappings) are specified and created at runtime
  ➔ Base for graceful degradation
  ➔ Base for data-dependent dynamic schedules
Callable schedules

- Complete schedules can be inserted into other schedules. These schedules may
  - be provided by a different application
  - run on other processing nodes
  - run on other operating systems

⇒ Schedules become reusable parallel components
Callable schedules

User App #1
Striped File System

User App #1
Striped File System

User App #1
Striped File System

User App #2
Striped File System

User App #2
Striped File System

User App #2
Striped File System
Execution model

- A DPS kernel daemon is running on all participating machines
  - The kernel is responsible for starting applications (similar to rshd)
Implementation

- DPS is implemented as a C++ library
  - No language extensions, no preprocessor
- All DPS constructs are represented by C++ classes
- Applications are developed by deriving custom objects from the provided base classes
- Cross-platform, supports heterogeneous execution environments
Operations

- Custom operations derive from provided base classes (\textit{SplitOperation}, \textit{LeafOperation}, \ldots)
- Developer overloads \textit{execute} method to provide custom behavior
- Output data objects are sent with \textit{postToken}
- Classes are templated for the input and output token types to ensure graph coherence and type checking
class Split : public SplitOperation
<MainThread,tv1(SplitInToken),tv1(SplitOutToken)>
{
public:
  void execute(SplitInToken *in)
  {
    for(Int32 i=0;i<IMAGE_SIZE_Y;i+=32)
    {
      postToken(new SplitOutToken(i,32,j));
    }
    dump(0,"Finished split, %d tokens posted",j);
  }
  IDENTIFYOPERATION(Split);
};
Threads

• Derives from provided *Thread* class
• One instance of this class is created for every thread in a collection
• This class can contain data elements
  – They are preserved as long as the thread exists
  – Operations executing on a thread can access thread local variables using member *thread*
class ProcessThread : public Thread
{
public:
    Buffer<UInt8> world[2];
    Int32 firstLine;
    Int32 lineCount;
    Int32 active;

    IDENTIFY(ProcessThread);
};
Routing

- Routing functions derive from base class Route
- Overloaded method route selects target thread in collection based on input data object
- DPS also provides built-in routing functions with special capabilities
Routing

Class RoundRobinRoute : public Route<SplitOutToken>
{
  public:
    Int32 route(SplitOutToken *currentToken) {
      return currentToken->target%threadCount();
    }
    IDENTIFY(RoundRobinRoute);
};
Data Objects

• Derive from base class *Token*
• They are user-defined C++ objects
• Serialization and deserialization is automatic
  – Data Objects are not serialized when passed to another thread on a local machine (SMP optimization)
• Three serialization models are provided:
  – Simple, Complex, and Universal
Data Objects (Simple)

class SplitOutToken : public SimpleToken
{
    public:
        Int32 sizeX;
        Int32 sizeY;

    SplitOutToken() {}

    IDENTIFY(SplitOutToken);
};

• The class is serialized by a simple memory copy
  - Unsuitable for heterogeneous environments
  - Unsuitable for complex data
Data Objects (Complex)

```cpp
class MergeInToken : public ComplexToken
{
public:
    CT<Int32> sizeX;
    CT<Int32> sizeY;
    Buffer<UInt8> pixels;
    CT<Int32> scanLine;
    CT<Int32> lineCount;

    MergeInToken() {}

    IDENTIFY(MergeInToken);
};
```

- All used types derive from a common base class (Object)
- The class is serialized by walking through all its members
class MergeInToken : public Token
{
    CLASSDEF(MergeInToken)
    BASECLASS(Token)
    MEMBERS
        ITEM(Int32,sizeX)
        ITEM(Int32,sizeY)
        ITEM(Buffer<UInt8>,pixels)
        ITEM(Int32,scanLine)
        ITEM(Int32,lineCount)
    CLASSEND;

    MergeInToken() {} };

- Can contain anything, derive from anything, be a template, etc.
- Provides a fully static generic reflection mechanism for class contents
- Makes no assumptions about compiler ABI
Data Object Serialization

- Complex and Universal serializers use knowledge about data to ensure correct conversion in heterogeneous environments.
- Complex structures such as circular lists or trees can be serialized.
- The Universal serializer can also read and write complex data formats, such as XML.
STL Support

- Universal serializer fully supports STL containers and strings

```cpp
class SomeToken : public Token {
{
    CLASSTYPE(MergeInToken)
    BASECLASS (Token)
    MEMBERS
        ITEM (std::string, name)
        ITEM (std::vector<int>, sizes)
        ITEM (std::map<int, std::string>, aMap)
    CLASSEND;
};
```

- Requires compiler support for partial template specialization
Thread collections

• Creation of an abstract thread collection

\[ \text{Ptr\textless ThreadCollection\textless ComputeThread\textgreater} > \]
\[ \text{computeThreads} = \text{new} \]
\[ \text{ThreadCollection\textless ComputeThread\textgreater} ("proc"); \]

• Mapping of thread collection to nodes
  - Mapping is specified as a string listing nodes to use for the threads

\[ \text{computeThreads}\rightarrow\text{map} ("\text{nodeA}\ast2 \text{ nodeB}"); \]
Graph construction

FlowgraphNode<Split,MainRoute> s(mtc);
FlowgraphNode<Process1,RoundRobinRoute> p1(ptc);
FlowgraphNode<Process2,RoundRobinRoute> p2(ptc);
FlowgraphNode<Merge,MainRoute> m(mtc);

FlowgraphBuilder gb;
gb = s >> p1 >> m;
gb += s >> p2 >> m;

Ptr<Flowgraph> g=new Flowgraph(gb,"myGraph");
Data transfer performance

DPS structured data vs. raw sockets

Data transfer throughput

Throughput [MB/s]

Single transfer data size [bytes]

DPS

Sockets
Overlapping performance

Matrix multiplication

- Communication/Computation ratio can be changed by modifying block size

![Chart showing time (ms) vs block size for 2, 3, and 4 nodes with non-overlapped and overlapped performance.

- For 2 nodes, the overlapped performance is 32.1% faster.
- For 3 nodes, the overlapped performance is 35.6% faster.

Block size: 256, 128, 64, 32
Non-Overlapped
Overlapped
Application examples

Game of life
- Requires neighborhood exchanges
- Uses thread local storage for world state

Neighbor exchange

Computation
Improved implementation

- Parallelize data exchange and computation of non-boundary cells
Performance

Speedup of the game of life

- Imp 400x400
- Std 400x400
- Imp 4000x400
- Std 4000x400
- Imp 4000x4000
- Std 4000x4000

Number of nodes

Speedup
Interapplication communication

Client application

Game of Life

Display graph

Collection graph

Computation graph
LU decomposition

\[ A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} r \]

\[ A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & B \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & X \end{bmatrix} \cdot \begin{bmatrix} U_{11} & T_{12} \\ 0 & Y \end{bmatrix} \]

\[ \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \cdot U_{11} \]

\[ A_{12} = L_{11} \cdot T_{12} \]

Compute LU factorization of block, stream out trsm requests.

Compute trsm, perform row flipping, return notification.
LU decomposition

\[ B = L_{21} \cdot T_{12} + X \cdot Y \]
\[ A' = X \cdot Y = B - L_{21} \cdot T_{12} \]

- Collect notifications, stream out multiplication orders.
- Multiply, subtract and store result, send notification.
- As soon as first column is complete, perform LU factorization. Stream out trsm while other columns complete the multiplication. Send row flip to previous columns to adjust for pivoting.
LU decomposition

\[
\begin{array}{c}
\mathbf{A_{11}} \\
\mathbf{A_{21}}
\end{array}
\]
LU decomposition
LU decomposition

\[
\begin{array}{ccc}
A_{12} & T_{12} & T_{12} \\
L_{21} & & \\
L_{21} & & \\
L_{21} & & 
\end{array}
\]
LU decomposition
LU decomposition
LU decomposition
LU decomposition

LU decomposition diagram with operations and matrices.
LU decomposition
LU decomposition

\[ \begin{align*}
\text{lu} & \rightarrow \text{trsm} \rightarrow \text{Mul} \rightarrow 9x \rightarrow \text{lu} \\
\text{lu} & \rightarrow \text{trsm} \rightarrow \text{Mul} \rightarrow \text{trsm} \rightarrow \text{Mul} \rightarrow \text{trsm} \rightarrow \text{Mul} \\
\text{lu} & \rightarrow \text{xchg} \rightarrow \text{Mul} \\
\text{lu} & \rightarrow \text{xchg} \rightarrow \text{Mul} \\
\text{lu} & \rightarrow \text{Mul} \\
\text{lu} & \rightarrow \text{xchg} \rightarrow \text{Mul} \\
\text{lu} & \rightarrow \text{xchg} \\
\text{lu} & \rightarrow \text{xchg} \\
\end{align*} \]
LU decomposition
LU decomposition
LU decomposition
Multiplication graph

- **Mul**
  \[ \text{Mul} \]
  = Split
  Collect operands
  Multiply
  Store result

- P1 \quad P2 \quad P3 \quad P4
Multiplication graph

\[ \text{Mul} = \begin{array}{c}
\text{Split} \\
\text{Collect operands} \\
\text{Multiply} \\
\text{Store result}
\end{array} \]

P1 P2 P3 P4
Multiplication graph

\[ \text{Mul} = \begin{array}{cccc}
\text{Split} & \text{Collect operands} & \text{Multiply} & \text{Store result} \\
\text{P1} & \text{P2} & \text{P3} & \text{P4}
\end{array} \]
Multiplication graph

\[
\text{Mul} = \text{Split} \quad \text{Collect operands} \quad \text{Multiply} \quad \text{Store result}
\]
LU Decomposition

Graph showing the speedup of LU decomposition with respect to the number of nodes. The graph compares pipelined and non-pipelined methods.
Conclusion

• DPS Characteristics
  - Dynamic construction of parallel schedules
  - Automatic pipelining helps hiding communication and I/O times
  - Deadlock-free programming model
  - Easy to understand/use
  - Support for multithreading on shared memory multiprocessors
  - Flow control/load balancing primitives
  - Heterogeneous execution environments
Conclusion

- Easy to install and use
- Potential of dynamic schedules for future research:
  - Reusable parallel components
  - Graceful degradation
  - Runtime reconfiguration of parallel programs
- DPS is available on the web: http://dps.epfl.ch