[simula . research laboratory]

UNIVERSITY OF OSLO

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Paul B. Beskow, Håvard Espeland, Håkon K. Stensland, Preben N. Olsen, Ståle Kristoffersen
Espen A. Kristiansen, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway and IFI, University of Oslo, Norway

P2G is a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing, where we seek to scale transparently with the available resources. P2G consists of a compiler and run-time that analyzes dependencies dynamically and merges or splits kernel instances based on resouce availability and performance monitoring.

KERNEL LANGUAGE

fetch <field F> (age A) [index X] ... [index Z]

Extract a slice of data from a field

store <field F> (age A) [index X] ... [index Z]

Store a slice of data to a field

P2G Kernel language Example

Field definitions:

```
0  int32[] m_data age;
1  int32[] p_data age;
```

Kernel definitions:

```
0  init:
1    local int32[] values;
2
3    %{
4      int i = 0;
5      for( ;i < 5; ++i )
6      {
7        put( values, i+10, i );
8      }
9    %}
10
11   store m_data(0) = values;
12
13
```

```
0  mul2:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = m_data(a)[x]
6
7    %{
8      value *= 2;
9    %}
10
11   store p_data(a)[x] = value;
12
13
```

```
0  plus5:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = p_data(a)[x]
6
7    %{
8      value += 5;
9    %}
10
11   store m_data(a+1)[x] = value;
12
13
```

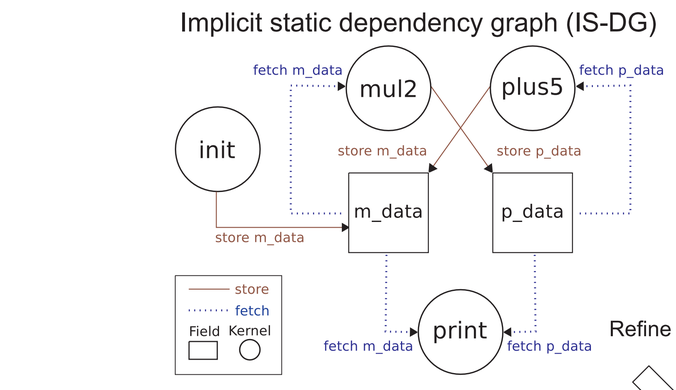
```
0  print:
1    age a;
2    local int32[] p, m;
3
4    fetch p = p_data(a)
5    fetch m = m_data(a)
6
7    %{
8      for(int i=0; i < extent(p, 0))
9      {
10       cout << "p: " << get(p, i);
11       cout << "m: " << get(m, i);
12     }
13   %}
```

C++ Equivalent

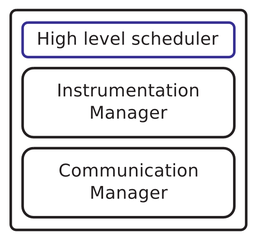
```
void print( int * data, int num )
{
  for( int i = 0; i < num; ++i)
    std::cout << data[i] << " ";
  std::cout << std::endl;
}

int main()
{
  int data[] = { 10, 11, 12, 13, 14 };
  int num = (sizeof data/sizeof *data);
  print( data, num );
  while(true)
    {
      for( int i = 0; i < num; ++i)
        {
          data[i] *= 2;
        }
      print( data, num );
      for( int i = 0; i < num; ++i)
        {
          data[i] += 5;
        }
      print( data, num );
    }
  return 0;
}
```

P2G ARCHITECTURE



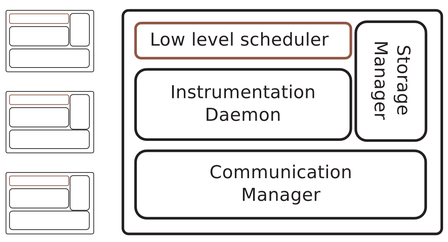
Receive workload



Master node

Partition graph

Refine graph

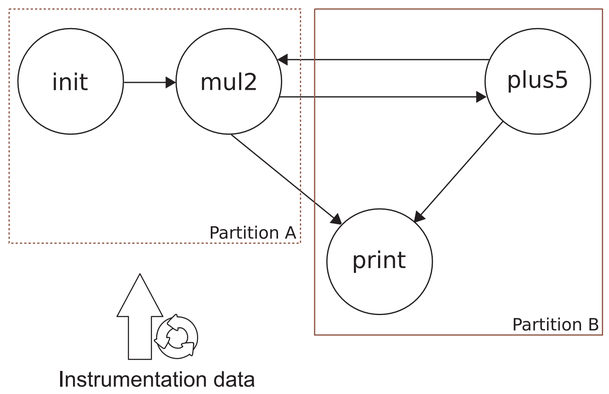


Execution node

Distribute full/partial workload to execution nodes

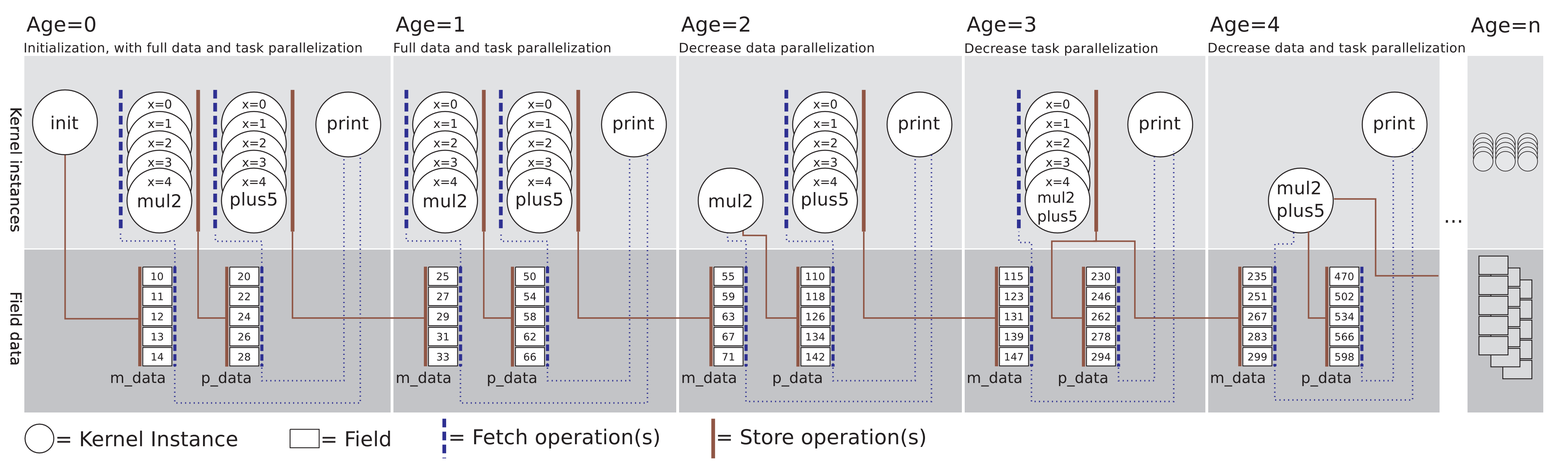
Execute workload

Refined implicit static dependency graph (RIS-DG)



Instrumentation data

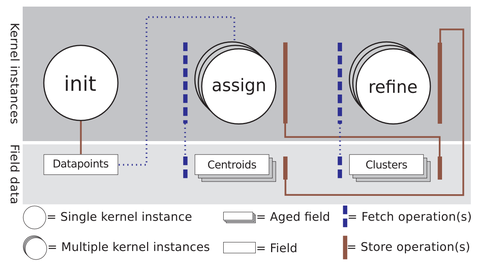
Dynamically created directed acyclic graph (DC-DAG)



K-MEANS CLUSTERING

Workload description

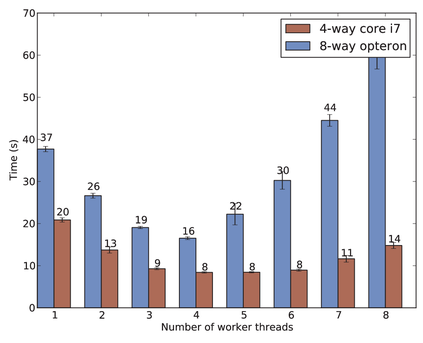
K-means clustering is an iterative algorithm for cluster analysis, which aims to partition *n* datapoints into *k* clusters in which each datapoint belongs to the cluster with the nearest mean.



The **init** kernel generates *n* datapoints, selecting *k* initial centroids. This results in *n* **assign** kernel instances per age, each fetching a single datapoint and the set of centroids for that age. It then calculates which of these centroids the point is closest to and stores the point in the cluster of that centroid (for the given age). *k* **refine** kernels are then dispatched (per age), each calculating the new mean of that cluster of datapoints, and then stores the result in the Centroids field. This process is then repeated until the algorithm converges.

Evaluation

The *k*-means workload was run on a generated dataset of *n*=2000 with *k*=100. The algorithm was not run until convergence, but stopped after 10 iterations, as the point of convergence is not deterministic.



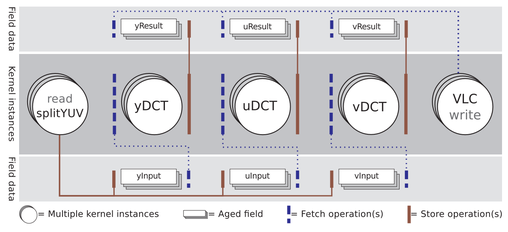
Microbenchmark

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 zs	9829.00 zs
assign	2024251	4.07 zs	6.95 zs
refine	1000	3.21 zs	92.91 zs
print	11	1.09 zs	379.36 zs

MOTION JPEG

Workload description

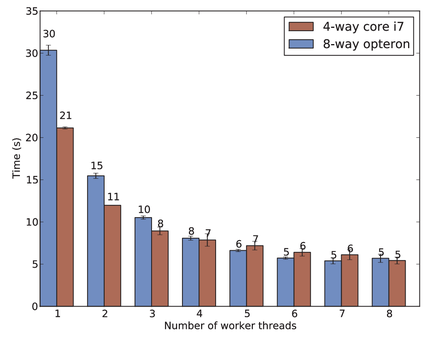
Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images.



One **read/splitYUV** kernel instance is dispatched per frame read. It splits the image into its Y, U and V components and stores this to its corresponding fields. Then one kernel per macro block is dispatced, with CIF resolution (352*288) this equals 1584 **yDCT** kernels, 396 **uDCT** and **vDCT** kernels. These kernels then calculate the discreet cosine transform and do quantization, storing their results to their corresponding fields. Then one **VLC/write** kernel per frame is dispatched to perform variable length coding and writing the buffer to disk.

Evaluation

The MJPEG workload was run on the standard test sequence Foreman encoded in CIF resolution. We limited the workload to process 50 frames of video. A single-thread native implementation completed in ~30 sec on the Opteron and ~19 sec on the Core i7.



Microbenchmark

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 zs	18.00 zs
read/splitYUV	51	35.50 zs	1641.57 zs
yDCT	80784	3.07 zs	170.30 zs
uDCT	20196	3.14 zs	170.24 zs
vDCT	20196	3.15 zs	170.58 zs
VLC/write	51	3.09 zs	2160.71 zs

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Paul B. Beskow*, Håvard Espeland*, Håkon K. Stensland*, Preben N. Olsen*,
Ståle Kristoffersen*, Espen A. Kristiansen*, Carsten Griwodz, Pål Halvorsen

*Student author

Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{paulbb, haavares, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

1. INTRODUCTION

As the number of multimedia services grows, so does the computational demands on multimedia data processing. New multi-core hardware architectures provide the required resources, however, parallel, distributed applications are much harder to write than sequential programs. Large processing frameworks like Google's MapReduce [1] and Microsoft's Dryad [2] are steps in the right direction, but they are targeted towards batch processing. As such, we present *P2G*, which is a framework designed to integrate concepts from modern batch processing frameworks into the world of real-time multimedia processing. With P2G we seek to scale transparently with the available resources (following the cloud computing paradigm) and to support heterogenous computing resources, such as GPU processing cores. The idea is to encourage the application developer to express as fine a granularity as possible along two axes, data and functional parallelism, where many of the existing systems sacrifice flexibility in one axis to accommodate for the other, e.g., MapReduce has no flexibility in the functional domain, but allows for fine-grained parallelism in the data domain. In P2G, functional blocks are formulated as kernels that operate on slices of multi-dimensional fields. As such, the fields, used to storing of the multimedia data, are used to express data decomposition. The write-once semantics of the fields provide the needed boundaries and barriers for functional decomposition to exist in our run-time and ensures deterministic output. P2G has intrinsic support for deadlines, and the compiler and run-time analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring. We have implemented a prototype of a P2G execution node, with MJPEG as a primary workload. As such, we intend to demonstrate an operating execution node of P2G at the conference. We will also present results from running tests on our execution node using MJPEG and K-means, where we are able to show that P2G scales with the number of cores available.

2. ARCHITECTURE

As shown in figure 1, P2G consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (i.e., multi-core, GPU, etc) to the master node, which combines this information to form a global topology of available resources. As such, the global topology can change during run-time as execution nodes can be dynamically added and removed to accommodate for changes in the global load.

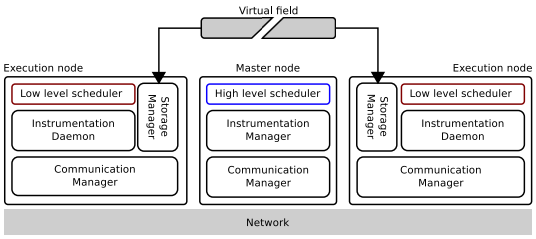


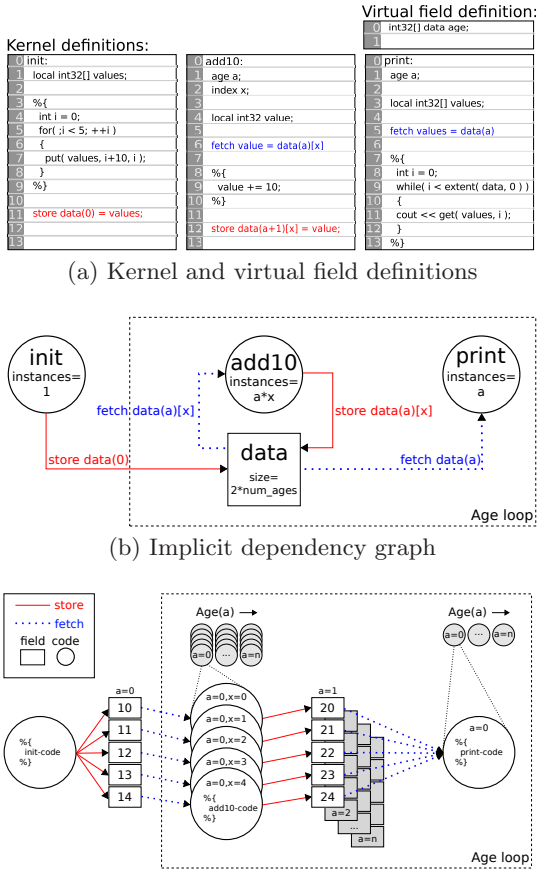
Figure 1: Overview of nodes in the P2G system.

As the master node receives workloads, it use its high-level scheduler to determine which execution nodes to delegate partial or complete parts of the workload to. This process can be achieved in a number of ways. However, as a workload in P2G forms an implict dependency graph based on its store and fetch operations to virtual fields, the high-level scheduler can utilize graph partitioning algorithms, or similar, to map such an implicit dependency graph to the global topology. The utilization of available resources is thus maximized.

When an implicit graph is split across multiple execution nodes, communication is achieved through an event-based, distributed publish-subscribe model. For every input, these subscriptions are deterministically derived from the code and the high-level schedulers partitioning decisions. The subscriptions also make it possible to establish direct communication links between the interacting execution nodes.

P2G uses a low-level scheduler at each execution node to maximize the local scheduling decisions, i.e., the low-level scheduler can decide to combine functional and data decomposition to minimize overhead. During run-time the master node will collect statistics on resource usage from all execution nodes, which all run an instrumentation daemon to acquire this information. The master node can then combine this run-time instrumentation data with the implicit dependency graph derived from the source code and the global topology to make continuous refinements to the high-level scheduling decisions. As such, P2G relies on its combincation of a high-level scheduler, low-level schedulers, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

As seen in figure 2, P2G provides a kernel language for the programmer to write their application in, which they do



(a) Kernel and virtual field definitions

(b) Implicit dependency graph

(c) Kernel and field instances

Figure 2: P2G programming model

by writing isolated, sequential pieces of code called *kernels*. Kernels operate on slices of *fields* through *fetch* and *store* operations and have native code embedded within them. In the model we encourage the programmer to specify the inheret parallelism in their application in as fine a granularity as possible in the domains of functional and data decomposition, without needing to sacrifice the one for the other.

The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to *fetch* slices of data in as fine granularity as possible. The write-once semantics of the fields provide deterministic output, though not necessarily deterministic execution of individual kernels. Given write-once semantics, iteration is supported in P2G by introducing the concept of aging, as seen in figure 2(b), where storing and fetching to the same field position, at different ages, makes it possible to form loops. The write-once semantics also provide natural boundaries and barriers for functional decomposition, as the low-level scheduler can analyze the dependencies of a kernel instance to determine if it is ready for execution. Furthermore, the compiler and the run-time, can analyze dependencies dynamically and merge or split kernels based on resource availability and performance monitoring.

Given a workload specified using the P2G kernel language, P2G is designed to compile the source code for a number of heterogenous architectures, though it currently only does so for the x86 architecture. P2G can then distribute this workload across the resources available to it.

At the time of writing, P2G consists of what we call an execution node, which is capable of executing entire workloads on a single x86 multi-core node. As such, the high-level scheduler and distribution mechanisms are not yet implemented, though the work is well under way.

3. WORKLOAD

We have implemented a few simple workloads used in multimedia processing to test the prototype implementation, here we will focus on our Motion JPEG implementation.

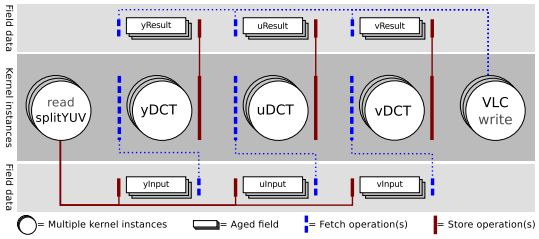


Figure 3: Overview of the P2G MJPEG encoding process

The *read + splitYUV* kernel reads the YUV-input video and stores the data in three fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 3, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC + write* kernel does variable length coding and store the MJPEG bit-stream to disk.

4. POSTER & DEMO

In this poster/demo, we will explain and discuss the P2G ideas for multimedia processing with deadlines. Additionally, we accompany the poster with a demo of several well known multimedia workloads and show the entire application development and processing pipeline, i.e, the code in kernel language, the compilation with parallel code generation and the processing automatically distributing the multimedia load to the available processing cores.

5. REFERENCES

[1] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI* (2004), pp. 10–10.

[2] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys* (New York, NY, USA, 2007), ACM, pp. 59–72.