Pattern-based Parallel Edge Preserving Algorithm for Salt-and-Pepper Image Denoising

Marco Aldinucci - Uni. of Torino, Italy, HPCAC University Award 2011 Winner
Maurizio Drocco - Uni. of Torino, Italy, supported by HPCAC University Award 2011
Massimo Torquati and Marco Danelutto - Uni. Pisa, Italy
Concetto Spampinato - Uni. of Catania, Italy
Massimiliano Meneghin - IBM Research, Ireland
Peter Kilpatrick - Queen’s Uni. Belfast, U.K.
ParaPhrase

Parallel Patterns for Heterogeneous Multicore Systems

EC-STREP FP7

Concurrency and multi-core, the theoretical background

a personal perspective

FastFlow

A programming model (and a library)

Fast (<10 nS) core-to-core lock-free messaging

Supporting multi-core and accelerators

Testbed: a novel two-phase image/video denoiser

clean salt-and-pepper noise up to 90%

reconstruction quality at 50% close to jpeg compression

demo
ParaPhrase: Parallel Patterns for Heterogeneous Multicore Systems

- 3 Year targeted research project (FP7 STREP)
  - Runs from 1/10/11 to 30/9/14
  - Funded by objective 3.4, “Computing Systems”
  - Project Number ICT-2011-288570

- 9 partners from five countries
- €3.5M budget, €2.6M EU contribution
- Coordinated by the U. of St Andrews, Scotland, UK
## Project Consortium

<table>
<thead>
<tr>
<th>Project partners</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of St Andrews</td>
<td>UK</td>
</tr>
<tr>
<td>Robert Gordon University</td>
<td>UK</td>
</tr>
<tr>
<td><strong>Mellanox Technologies Ltd.</strong></td>
<td>Israel</td>
</tr>
<tr>
<td>Software Competence Center</td>
<td>Austria</td>
</tr>
<tr>
<td>Hagenberg</td>
<td></td>
</tr>
<tr>
<td>Erlang Solutions Ltd</td>
<td>UK</td>
</tr>
<tr>
<td>University of Pisa</td>
<td>Italy</td>
</tr>
<tr>
<td>High Performance Computing Center</td>
<td>Germany</td>
</tr>
<tr>
<td>Stuttgart</td>
<td></td>
</tr>
<tr>
<td><strong>University of Torino</strong></td>
<td>Italy</td>
</tr>
<tr>
<td>Queen’s University Belfast</td>
<td>UK</td>
</tr>
</tbody>
</table>
Multi-cores: programming issues

✴ We can muddle through on 2-16 cores
✧ modified sequential code may work
✧ we may be able to use multiple programs to soak up cores
✧ BUT larger systems are much more challenging

✴ Fundamentally, programmers must learn to “think parallel”
✧ this requires new high-level programming constructs
✧ you cannot program effectively while worrying about deadlocks etc
  • they must be eliminated from the design!
✧ you cannot program effectively while fiddling with communication etc
  • this needs to be packaged/abstracted!
The need of a *programming model*

- Applications programmers must be systems programmers
  - insufficient assistance with abstraction, too much complexity to manage
- Difficult/impossible to scale, unless the problem is simple
- Future multi/many-core
  - Probably not just scaled versions of today’s multi/many-core
    - Highly heterogeneous: message-passing, shared-memory cc-NUMA, variable cache consistency, specialized cores (XML, crypto, ...), soft-cores (FPGA), ...
- The approaches provide libraries
  - they need to provide abstractions, i.e. *programming models*
The Implications for Programming

- We must program heterogeneous systems in an integrated way
- It will be impossible to program each kind of core differently
- It will be impossible to take static decisions about placement etc.
ParaPhrase Aims

Overall, we aim to produce a new pattern-based approach to programming parallel systems.

1. develop high-level design and implementation patterns that are capable of easily exposing useful parallelism on heterogeneous multicore/manycore systems.

2. develop new dynamic mechanisms to support adaptivity for heterogeneous multicore/manycore systems

3. verify that these patterns and adaptivity mechanisms can be used easily and effectively to develop a wide range of real-world applications
Concurrency and multi-core theoretical background: a personal perspective
Nowadays

- E.g. Intel Sandybridge, AMD Opteron
  - cache-coherent
  - 10 core per socket (20 contexts)
  - cc-NUMA (as matter of a fact)

- NVidia/AMD GPGPU/Hybrid
  - SIMD, no global synch
  - performance only with proper and not automatic memory hierarchy management

- Intel MIC CPU/GPGPU
  - ring-based interconnection, variable coherency
  - apparently even more NUMA

- IBM powerEN
  - general purpose cores
  - specialised cores, soft cores?
From programming/tuning viewpoint ... the simplest is already too complex ...

- Exploit cache coherence
- Memory fences are expensive
  - Increasing core count will make it worse
- Fine-grained parallelism is hard to achieve
  - I/O bound problems, High-throughput, Streaming, Irregular DP problems
  - Automatic and assisted parallelisation solves uniform & easy cases
    - OpenMP, Par4all, ...
- From programming model viewpoint, SIMD/GPGPU (maybe) worsen the scenario
  - Well-understood programming model
  - Atomic ops in memory (i.e. fences) are still needed
  - Not everything can be described with do independent (a.k.a. map)
Micro-benchmarks: farm of tasks

Used to implement: parameter sweeping, master-worker, etc.

```c
void Emitter() {
    for (i = 0; i < streamLen; ++i) {
        task = create_task();
        queue = SELECT_WORKER_QUEUE();
        queue->PUSH(task);
    }
}

void Worker() {
    while (!end_of_stream) {
        myqueue->POP(&task);
        do_work(task);
    }
}

int main() {
    spawn_thread(Emitter);
    for (i = 0; i < nworkers; ++i) {
        spawn_thread(Worker);
    }
    wait_end();
}
```

![Diagram showing the flow of tasks between an Emitter (E) and multiple Workers (W1, W2, ..., Wn) that eventually feed into a Collector (C).]
Task farm with POSIX lock/unlock

Speedup vs. Number of Cores

Ideal: 8 - 0.5 μS, 5 μS, 50 μS

average execution time per task
Can we avoid locks?

- Under relaxed memory models, using CAS/atomic ops
  - “lock-free” data structures
  - they perform better than lock-based
  - they fence the memory and pay cache coherency reconciliation overhead
  - in GPUs ...
    - CAS/atomic ... you have to go to the global memory
Lock vs CAS at fine grain (0.5 μS)
Re-starting from the basics

✴ Reducing the problem to the bare bones

✦ Producer-Consumer model (streaming)

✦ Directly control thread blocking using non-blocking synchronisations

✦ Directly design the “data channel”
  • Having clear how data move in the whole memory hierarchy

✴ Restarting from the FIFO queue
Interaction models: theoretical background (in a nutshell)

✴ low-level synchronisation in the shared memory model
  ✦ Mutual Exclusion (mutex)
    • typically used as basic building block of synchronisations
  ✦ Producer Consumer

✴ they are not equally demanding
  ✦ Mutual Exclusion is inherently more complex since requires deadlock-freedom
    • require interlocked ops (CAS, ...), that induces memory fences, thus cache invalidation
    • Dekker and Bakery requires Sequential Consistency (++)
  ✦ Producer Consumer is a cooperative (non cyclic) process
Recap: coherence and consistency

Memory/Cache Coherence

- Deal with multiple replicas of the same location in different caches

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{write}(A,3))</td>
<td>(\text{write}(A,1))</td>
</tr>
<tr>
<td>(\text{read}(A,?))</td>
<td>(\text{write}(A,1))</td>
</tr>
</tbody>
</table>

Memory Consistency

- Deal with the ordering in which writes and reads at different locations take effect in memory (issued by either the same or different processors/cores)
FastFlow SPSC queues

lifting_nonblocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

lifting_nonblocking(data) {
    if (head == tail) {
        return EWOULDBLOCK;
    }
    data = buffer[tail];
    tail = NEXT(tail);
    return 0;
}

* Proved to be correct under SC
* doesn’t work under weaker models
  • TSO, e.g. x86
  • WO, e.g. PPC, ARM

Lamport FIFO - 1983
FastFlow SPSC queues

**Lamport FIFO - 1983**

```c
push_nonbocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    if (head == tail) {
        return EWOULDBLOCK;
    }
    data = buffer[tail];
    tail = NEXT(tail);
    return 0;
}
```

**FastFlow FIFO**

derived from P1C1 (Higham and Kavalsh, ISPA 1997)

```c
push_nonbocking(data) {
    if (NULL != buffer[head]) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    data = buffer[tail];
    if (NULL == data) {
        return EWOULDBLOCK;
    }
    buffer[tail] = NULL;
    tail = NEXT(tail);
    return 0;
}
```

*(WMB)*

For any model weaker than TSO
FastFlow SPSC queues

Lamport FIFO - 1983

```c
push_nonblocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    if (head == tail) {
        return EWOULDBLOCK;
    }
    data = buffer[tail];
    tail = NEXT(tail);
    return 0;
}
```

FastFlow FIFO

```c
push_nonblocking(data) {
    if (NULL != buffer[head]) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    data = buffer[tail];
    if (NULL == data) {
        return EWOULDBLOCK;
    }
    buffer[tail] = NULL;
    tail = NEXT(tail);
    return 0;
}
```

head and tail are mutually invalidated by producer and consumer. 1 cache miss every push and pop (at least)

For any model weaker than TSO, producer read/write head, consumer read/write tail, no misses

excluding “true” deps

extended domain on void *
Lock vs CAS vs SPSC FastFlow (50 μS)
Lock vs CAS vs SPSC FastFlow (5 μS)
Lock vs CAS vs SPSC FastFlow (0.5 μS)
Medium grain (5 μS workload)
Fine grain (0.5 μS workload)
Producer-Cons with FF queue: latency

4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

DK Panda said today
MPI is ~190 ns

same socket different cores different

same socket same core different contexts

different sockets
Wait-free fence-free unbound dynamic queue (uSPSC)

```c
1. int size = N; // SPSC size
2. bool push(void* data) {
3.   if (buf_w->full()) {
4.     SPSC* t = pool.next_w();
5.     if (!t) return false;
6.   }
7.   buf_w = t;
8.   buf_w->push(data);
9.   return true;
10. }
11. bool pop(void** data) {
12.   if (buf_r->empty()) {
13.     if (buf_r == buf_w) return false;
14.     if (buf_r->empty()) {
15.       SPSC* tmp = pool.next_r();
16.       if (tmp) {
17.         pool.release(buf_r);
18.         buf_r = tmp;
19.       }
20.     }
21.   }
22.   return buf_r->pop(data);
23. }
24. struct Pool {
25.   dSPSC inuse;
26.   SPSC cache;
27.   SPSC* next_w() {
28.     SPSC* buf;
29.     if (!cache.pop(&buf))
30.       buf = allocateSPSC(size);
31.     inuse.push(buf);
32.     return buf;
33.   }
34.   SPSC* next_r() {
35.     SPSC* buf;
36.     return (inuse.pop(&buf)? buf : NULL);
37.   }
38.   void release(SPSC* buf) {
39.     buf->reset(); // reset pread and pwrite
40.     if (!cache.push(buf))
41.       deallocateSPSC(buf);
42.   }
43. }
```

Fig. 3: Unbounded wait-free uSPSC queue implementation.

Bandwidth test (with uSPSC)

http://www.1024cores.net/home/technologies/fastflow

4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

<35 ns irrespectively of the mapping

Linked list + circular buffer
FastFlow queue (our result)

12x faster

Linked list with pooling
Opt Michael-Scott queue (our result)

20x faster

Linked list w/ dyn alloc
Michael-Scott queue well-known ~ 400 citations

Throughput (msg/s)

n. threads

uSPSC
dSPSC
dSPSC no cache

Speedup

n. threads

1  8 16 24 32 40 48 56 64

50K 100K 150K 200K

50 150 250 350 450

nanoseconds

64 1024 8192

buffer size

Same core

Same socket

Different sockets

Mapping 1

Mapping 2

Mapping 3

Buffer size

64 1024 8192

nanoseconds

Mapping 1

Mapping 2

Mapping 3

Linked list + circular buffer
FastFlow queue (our result)

12x faster

Linked list with pooling
Opt Michael-Scott queue (our result)

20x faster

Linked list w/ dyn alloc
Michael-Scott queue well-known ~ 400 citations
Related Work: Lock-free, CAS-free, wait-free

**Single-Producer-Single-Consumer FIFO queues**
- Lamport et al. 1983 Trans. PLS (Sequential consistency only - passive)
- Higham and Kavalsh. 1997 ISPAN (P1C1 - TSO + proof - passive)
- Giacomoni et al. 2008 PPoPP (TSO + cache slipping - passive)

**Multiple-Producers-Multiple-Consumers FIFO queues**
- with CAS (two of them) - Michael and Scott (PODC96)
  - Also implemented in FastFlow, require deferred reclamation/hazard pointers to avoid ABA problem
- without CAS - passive ➔ Cannot be done
- without CAS - active ➔ FastFlow

Extending the taxonomy with locking algorithms is clearly useless ....
Why elaborating on queues?

Queue are the base of Producer-Consumer
- reference paradigm for data-centric and data-flow models
- orthogonal and complementary to Transactional Memories
  - that address competition
- can be used in both shared memory and message passing models

Queue are the primary (maybe the only) global synchronisation mechanism in current GPGPUs
- take a look to NVidia/CUDA programming manual
- efficient global synch can pave the way to a broader usage of GPUs
  - is it possible an atomic-free global memory queue on GPUs?
FastFlow
Lock-free and CAS-free?

✴ Mutex cannot be done
  Single-Producer-Single-Consumer (SPSC) can be done
  ✦ Producer-Consumer is inherently weaker with respect to Mutex
  ✦ It does require the cooperation of partners whereas Mutex does not

✴ Expressive enough to build a streaming (or dataflow) programming framework
  ✦ MPMC = SPSC + mediator threads

✴ But what about productivity at large scale?
  ✦ Write a program is defining a graph encoding true dependencies ... not really easy
FastFlow is based on producer-consumer

- Lock-free/fence-free non-blocking synchronisations
- C++ STL-like implementation
- thread-model agnostic (pthreads, QT, windows threads, ...)
- compliant with other synchronisation mechanisms in the business code (e.g. locks and semaphores)
Pattern-based approach: rationale

✶ Abstract parallelism exploitation pattern by parametric code
  ✦ E.g. higher order function, code factories, C++ templates, ...
  ✦ Can composed and nested as programming language constructs + offloading
  ✦ Stream and Data Parallel

✶ Platform independent
  ✦ Implementations on different multi/many-cores
  ✦ Support for hybrid architectures thanks to pattern compositionality

✶ Provide state-of-the-art, parametric implementation of each parallelism exploitation pattern
  ✦ With natural way of extending patterns, i.e. OO
  ✦ Functional (seq code) and tunable extra-functional (QoS) parameters
Patterns, their implementation, and their purpose

- **farm**
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA streams - automatic exploitation of asynch comm

- **pipeline**
  - on CPU - pipeline
  - on GPU - sequence of kernel calls or global mem synch

- **map**
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT - parallelism exploitation

- **reduce**
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT (reduction tree) - parallelism exploitation

- **D&C**
  - on CPU - master-worker with feedback - // exploitation
  - on GPU - working on it, maybe loop+farm
Composition via C++ template meta-programming

- CPU: Graph composition
- GPU: CUDA streams
- CPU+GPU: offloading

```cpp
farm{
pipe
}
```

```cpp
pipe(farm, farm)
```

```cpp
pipe(map, reduce)
```

```cpp
....
```
On my personal view of MPI ...

- I love it, I’ve using it for 15 years
  - as a target language for building compilers for high-level parallel languages

- MPI collective operations are good
- they are an example of patterns (a.k.a. skeletons)
  - even if not particularly high-level

- the problem are Send/Rcv
  - and the other over hundred (and counting) operations?
    - are they perceived as really useful from anybody?
  - they are on the language as well, and people use them

- one problem of “cars” are drivers (and crashes)
  - how much expertise do you need to write/tune/debug something barely decent?
At the bottom line, it is just abstracting & engineering well-known concepts.
Recap

✶ Patterns at the high-level
  ✦ Currently as C++ templates
  ✦ Set of patterns can be extended, semantics of patterns can be changed, complexity gracefully increase with semantic distance

✠ Used to generate cyclic streaming networks (of threads, ...)
  ✦ Graphs, describing true data dependencies. Can be composed and transformed as graphs
  ✦ Cyclic graphs need unbound queue
  ✦ Heterogeneous cores, thread affinity, memory affinity, NUMA, can be managed while mapping graph onto the metal
Applications (see http://di.unito.it/fastflow)

- Smith-Waterman (based on SWPS3/SSE2, ParCo 10)
  - Recently discovered it is used as baseline for FPGA design at it seems it is among the fastest SW solutions
- C4.5 (EMCL-PKDD 10, Concurrency prat. exp. submitted)
- StochKit-ff (HiBB 11)
- MonteCarlo sims & Gillespie’s biological sims (PDP 11, HiBB 11)
- Cholesky (PDP 12)
- nProbe-ff (part of nTop software, ParCo 11)
- pbzip-ff (Wiley book 12)
- nqueens, k-means, fibonacci, Mandelbrot QT, and many micro-benchmarks
- Two-phase edge-preserving denoiser (IEEE IPTA submitted)
Targeting multi cores by structured programming and data flow

M. Aldinucci, L. Anardu, M. Danelutto, P. Kilpatrick, M. Torquati

Dept. Computer Science, U of Toronto, Canada
Dept. Computer Science, Queen Mary, University of London, UK

Abstract

Data flow techniques have been around since the early days of compiler design. They were used to convert sequential language descriptions into parallel execution. Although the impact here was limited, data flow techniques have been identified as a candidate for efficient programming models on multi-core architectures. We have investigated the parallel implementation of an example program using the PLASMA library, while the compiler/runtime system schedules the instructions. We have used a programming approach supporting automatic parallelization on multi-core architectures. The results are encouraging.

Figure 10: $mdf^3$ vs PLASMA library. Cholesky factorization for a single 1024x1024 complex matrix (Intel Nehalem).
Decision Tree Growing and Pruning on General-Purpose Multicore

Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati

Abstract—The whole computer hardware industry embraced multicores. The extreme optimisation of sequential algorithms is then no longer sufficient to squeeze the real machine power, which can be only exploited via thread-level parallelism. Decision tree algorithms exhibit natural concurrency that makes them suitable to be parallelised. This paper presents an in-depth study of the parallelisation of an implementation of the C4.5 algorithm for multicore architectures. In addition to the tree growing phase, we cover also the so far unaddressed problem of parallelising the error-based pruning with grafting phase. We characterise elapsed time lower bounds for the forms of parallelisations adopted, and achieve close to optimal performances. Our implementation is based on the FastFlow para

Index Terms—Parallel clas:
Smith Waterman vs TBB vs OpenMP vs Cilk
Parallel Computing 2010
From users: Intel or AMD

🌟 We always test on both platforms

✦ ....

🌟 Assembler

✦ MONITOR/MWAIT instruction appear interesting for non-blocking synchronisations
  • accessible only in Ring 0, why?
  • IBM PowerEN ...

🌟 Memory bandwidth is the key

✦ but memory affinity tools appear to be quite primitive in both platforms
  • how data structures are related with access pattern?
  • we are trying to address the problem with patterns ...
A novel two-phase edge preserving parallel de-noising

Maurizio Drocco
University of Torino (MSc. student)
with the support of HPC AC university award
Two-phase denoising + Demo

Original
Baboon standard test image 1024x1024

Restored

10% impulsive noise

50% impulsive noise

90% impulsive noise

PNSR 43.29dB MAE 0.35

PNSR 32.75dB MAE 2.67

PNSR 23.4 MAE 11.21
Denoising explained

Map p in pixels
while (winsize<MAX)
  if (homogenous(p,winsize))
    winsize++;
  else if isImpluse(p) return NOISY;
return NOT_NOISY;

while !fixpoint
  map u in N (noisy pixels)
  new_u = value_that_minimize F(u);
  reduce(u in N,new_u in NEW_N, diff);

Adaptive
median filter

different pixels are
independent and can
be easily processed
in parallel
pixels are read-only

Iterative
variational method

answer to the question:
which is the greyscale level
that better “integrate” in
the surrounding
(i.e. keeps edges)
at each iteration an
approximation of restored
pixels is available

\[
F_d|_N(u) = \sum_{(i,j) \in N} [ |u_{i,j} - d_{i,j}| + \frac{\beta}{2} (S_1 + S_2)]
\]

\[
\varphi(t) = |t|^\alpha \text{ with } 1 < \alpha \leq 2
\]

\[
S_1 = \sum_{(m,n) \in V_{i,j} \cap N} 2 \cdot \varphi(u_{i,j} - d_{m,n})
\]

\[
S_2 = \sum_{(m,n) \in V_{i,j} \cap N^c} \varphi(u_{i,j} - u_{m,n})
\]
On the algorithm

✦ Adaptive median filters
  ✦ well-known
  ✦ smooth the image if used in rebuilding phase

✦ Variational methods
  ✦ edge-preserving, can be used for any kind of noise
  ✦ the formula is well-known, our algorithm is particularly fast
    • because we found several loop invariants in the process
  ✦ if used alone can destroy “good” pixels
  ✦ not used up to now because too slow
    • > 1 h on Matlab to rebuild a 256x256 image with 30% of noise

✦ Coupling them
  ✦ detect: increase the speed by reducing false positives
  ✦ denoise: guarantee a good restoration because “.touches” only noisy points
Offloading on soft (i.e. not used cores) and HW accelerators

- offloading
- onto other cores and accelerators
**Speedup (multi-core)**

![Graphs showing speedup](image)

- **variant: spd-ff, size: 4096x4096, n. cycles: 80**
  - ideal
  - 10% noisy
  - 50% noisy
  - 90% noisy

- **variant: spd-ff-border, size: 512x512, n. cycles: 60**
  - ideal
  - 10% noisy
  - 50% noisy
  - 90% noisy

4 sockets × 8 core × 2 contexts
Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2
Parallelism exploitation and patterns

✴ Application programmed high-level with FastFlow
  ✦ using Intel OpenCV
    ● be careful, it is not fully thread-safe, all non thread-safe operation should be managed by a single thread (i.e. main)
  ✦ using offloading support, i.e. wrap a pattern-based code into a soft or HW “accelerator”
    ● nice technique to port existing legacy code with very low effort
    ● support for “semi-automatic” loop streamization

✴ Let us understand how through an example
  ✦ Two-phase denoising - video version
#include <opencv/highgui.h>
#include <opencv/cv.h>

int main(int argc, char *argv[]) {
    CvCapture *capture;
    IplImage * frame,clean_frame;
    char key;
    vector<noisy_t> noisy;
    cvNamedWindow("Video", CV_WINDOW_AUTOSIZE);
    capture = cvCreateCameraCapture(CV_CAP_ANY);
    //capture = cvCreateFileCapture("/path/to/your/video/test.avi");
    while(true) {
        frame = cvQueryFrame(capture); // get a frame from device
        noisy = myDetect(frame); // detect noisy pixels
        clean_frame = myDenoise(frame,noisy); // denoise the frame
        cvShowImage("Video", clean_frame); // show the denoised frame
        key = cvWaitKey(100);
    }
    cvReleaseCapture(&capture);
    cvDestroyWindow("Video");
}
Denoising explained

Adaptive median filter

- Different pixels are independent and can be easily processed in parallel
- Pixels are read-only

Iterative variational method

- Answer to the question: which is the grayscale level that better "integrate" in the surrounding (i.e. keeps edges)
- At each iteration an approximation of restored pixels is available

\[ F_d|N(u) = \sum_{(i,j) \in N} |u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2) \]

\[ \varphi(t) = |t|^{\alpha} \text{ with } 1 < \alpha \leq 2 \]

\[ S_1 = \sum_{(m,n) \in V_i,j} 2 \cdot \varphi(u_{i,j} - d_{m,n}) \]

\[ S_2 = \sum_{(m,n) \in V_i,j} \varphi(u_{i,j} - u_{m,n}) \]

In the video case

- The two stages can be pipelined on m-core+m-core
- m-core+GPGPU
- GPGPU+GPGPU

and you haven’t to decide it at design time
do port the code
Single design - many implementations

Sequential

Parallel - $\text{Farm(Pipeline(myDetect, myDenoise))}$

Changing the structure does not require re-writing business code (gray)
Single design - many implementations

Sequential

Parallel - Pipeline(myDetect, Map(myDenoise))

Changing the structure does not require re-writing business code (gray)
Single design - many implementations

Sequential

Parallel - Farm(Pipeline(myDetect, Map(myDenoise)))

Changing the structure does not require re-writing business code (gray)
## FF multi-core vs FF hybrid

<table>
<thead>
<tr>
<th>noise</th>
<th>FF 32 cores Intel 4x8x2 2GHz</th>
<th>FF 8 cores (detect) + Tesla C2050 (denoise)</th>
<th>Seq Intel 4x8x2 2GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena 512x512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.8 s</td>
<td>1.9 s</td>
<td>32 s</td>
</tr>
<tr>
<td>50</td>
<td>6.5 s</td>
<td>2.3 s</td>
<td>162 s</td>
</tr>
<tr>
<td>90</td>
<td>10.9 s</td>
<td>2.8 s</td>
<td>290 s</td>
</tr>
<tr>
<td>Space 4096x4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>78 s</td>
<td>12 s</td>
<td>2093 s</td>
</tr>
<tr>
<td>50</td>
<td>373 s</td>
<td>46 s</td>
<td>10400 s</td>
</tr>
<tr>
<td>90</td>
<td>665 s</td>
<td>77 s</td>
<td>18571 s</td>
</tr>
</tbody>
</table>
Demo
Nondeterministic variants & Convergence speed

- **flat** (used in CUDA version)
  - use block halo, block size = 1
  - do independent
  - deterministic - slow convergence
  - easy CPU and GPU - can be linearized

- **border**
  - use block halo
  - do independent on tiles, do across within tiles
  - nondeterministic - fast convergence
  - easy on CPU and GPU - cannot be linearized

- **std** (used in multicore version)
  - don’t use a block halo
  - do independent on tiles, do across within tiles
  - nondeterministic - fast convergence
  - easy CPU and GPU - cannot be linearized

- **cluster**
  - do independent on tiles, do across within tiles
  - deterministic - fast convergence
  - easy on CPU, difficult on GPU - can be linearized
Conclusions

✶ FastFlow C++ pattern-based framework

✦ A tiny, lightweight & open research framework
  • 2 years old - 6K downloads - 30K contacts - x86/PPC/ARM(?) + Linux/Mac/Win
  • Adopted as one run-time technology in ParaPhrase
  • Laboratory to experiment new run-time solutions
    • GPGPU integration (working on), Infiniband RDMA integration (working on), HW blocking reads (thinking on), HW transactional mem (thinking on) ...
    • Stream-specific parallel memory allocator: faster than TBB (testing)

✦ A step forward in parallel programming models
  • Data-centric, focus on messaging and synchronization, thread model agnostic
  • support both message passing and shared memory

✧ High-level = performance & portability
  • Speedup starting from ~20 clock cycles workload on standard x86_64 (TBB ~ 50K)
  • FF/AVX Smith-Waterman among fastest existing SW solutions
  • Tested on dozen of apps, comparable or faster than TBB/OpenMP
Conclusions

✱ GPGPUs
  ✦ Needs high-level, CUDA/OpenCL too close to the metal
  ✦ Well integrate with functional style and higher order patterns

✱ Two-phase denoising
  ✦ Novel, fast and efficient
    • Works up to 95% or noise, comparable to jpeg on 50% of noise
  ✦ CPU/GPGPUs/Hybrid
  ✦ Edge-preserving restoration works also for other kinds of noise
    • Working on detection and color images
Thank you
FF-allocator (written in FF)

- Faster than posix, often faster than hoard and TBB
  - unpublished, but available on sourceforge
  - needs lot of comparative testing to be published
- Implements deferred deallocation to avoid ABA problem (i.e. garbage collection)

The graph is now cyclic and needs unbound queues to avoid deadlocks
On programming model

* Shared memory or message passing (data)
  + message passing (synchronisations)
    ✦ Graphs exactly describes the (true) data dependency pattern
    ✦ Additional synchronisations can be added (e.g. locks) in the user code

* Queue can pass pointers or data
  ✦ Passing data means copying it. A proper usage of allocator might significantly enhance locality. Copying non strictly needed data is overhead.
  ✦ The balance depends on the application. Should be studied more.
On programming model

- Shared memory or message passing (data) + message passing (synchronizations)
  - Graphs exactly describes the (true) data dependency pattern
  - Additional synchronizations can be added (e.g. locks) in the user code

- Queue can pass pointers or data
  - Passing data means copying it. A proper usage of allocator might significantly enhance locality. Copying non strictly needed data is overhead.

- The balance depends on the application. Should be studied more.
MPP

P3L
1991

Beowulf

SKiE
1997

OCamlP3L
1998

Shared
memory

Grid

Macro
Data Flow

GPGPUs

FastFlow
2009

Eskimo
2003

GCM
2008

ASSIST
2001

Lithium
2002

SKElab
2000

Muskel
2006

ASSISTant
2008

GCM
2008

Beowulf

Shared
memory

Macro
Data Flow

GPGPUs

FastFlow
2009

Eskimo
2003

GCM
2008

ASSIST
2001

Lithium
2002

SKElab
2000

Muskel
2006

ASSISTant
2008

GCM
2008