Main Benefits of Task Parallel Frameworks

Rafael Asenjo
Dept. of Computer Architecture
University of Málaga
http://www.ac.uma.es/~asenjo
Agenda

• Motivation
  – Task-parallel frameworks
  – Work Stealing

• Putting tasks to work
  – Pipeline
  – Wavefront
  – Heterogeneous

• Conclusions
Motivation

• Yes, we are in the multicore era!
  – Execution performance is …
    • Not a HW main problem any more
    • But a SW responsibility \(\rightarrow\) must parallelize the apps
  – Sequential program \(\rightarrow\) Slow program
  – SW developer \(\rightarrow\) Parallelizer
  – Think in parallel

• Needed:
  – New languages and tools to help developers
Parallel languages and tools

- **Parallel frameworks**
  - Distributed memory
    - Message passing: MPI, SHMEM, GASNet, …
  - Shared memory
    - Pthreads, OpenMP
    - Task frameworks:
      - Intel TBB, Cilk, Intel CnC, OpenMP 3.0, MS TPL, Java Conc.

- **Parallel Languages**
  - Partitioned Global Address Space (PGAS)
    - UPC, Co-array Fortran (CAF), Titanium (Parallel Java)
  - High Performance Computing Systems (HPCS)
    - Chapel (Cray), X10 (IBM), Fortress (Sun/Oracle)

- **Heterogeneous:** CUDA, OpenCL, OpenACC
Task vs. Threads

• **Tasks:** much lighter-weight than threads
  – Typically a function or class method
  – TBB: Task vs. thread
    • Linux: 18x faster to start and terminate a task
    • Windows: 100x faster
  – GCD (Apple’s Grand Central Dispatch)
    • Create a traditional thread: 100s instructions
    • Create a task: 15 instructions
  – In general: user level scheduler
    • The OS kernel does not see the tasks
    • The task scheduler’s mainly focus: performance
Work-stealing scheduler

Thread Pool: Work Stealing

Core 0

Local Queue

Task 0

Task 2

Task 1

Core 1

Local Queue

Worker Thread 1

Task 3

Task 4

Task 5

Core 2

Local Queue

Worker Thread 2

Task 6

Sept 13, 2012

HPC-AC. Rafael Asenjo.
Task scheduler advantages

• Automatic load balancing
  – With less overhead than work-sharing

• Without the OS scheduler restrictions
  – No OS scheduler preemption, no Round-Robin
  – Unfair scheduling: can sacrifices fairness for efficiency
  – Can be guided by the user
    • You don’t need to be “root” nor rely on RT priorities
    • Can be cache-conscious
  – Avoid oversubscription, and therefore:
    • Avoid context switching overhead
    • Avoid cache cooling penalty
    • Avoid lock preemption and convoying
The PARSEC suite

- Princeton Application Repository for Shared-Memory Computers (http://parsec.cs.princeton.edu)
  - Emerging multithreaded workloads
  - Diverse enough: RMS, streaming, multimedia, engineering, …
  - Originally implemented with Pthreads

<table>
<thead>
<tr>
<th>Program</th>
<th>Application Domain</th>
<th>Parallelization Model</th>
<th>Granularity</th>
<th>Working Set</th>
<th>Data Sharing</th>
<th>Usage Exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
<td>data-parallel</td>
<td>coarse</td>
<td>small</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>data-parallel</td>
<td>medium</td>
<td>medium</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>canneal</td>
<td>Engineering</td>
<td>unstructured</td>
<td>fine</td>
<td>unbounded</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>dedup</td>
<td>Enterprise Storage</td>
<td>pipeline</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>data-parallel</td>
<td>coarse</td>
<td>large</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>ferret</td>
<td>Similarity Search</td>
<td>pipeline</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation</td>
<td>data-parallel</td>
<td>fine</td>
<td>large</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>freqmine</td>
<td>Data Mining</td>
<td>data-parallel</td>
<td>medium</td>
<td>unbounded</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>streamcluster</td>
<td>Data Mining</td>
<td>data-parallel</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>swaptions</td>
<td>Financial Analysis</td>
<td>data-parallel</td>
<td>coarse</td>
<td>medium</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>vips</td>
<td>Media Processing</td>
<td>data-parallel</td>
<td>coarse</td>
<td>medium</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>x264</td>
<td>Media Processing</td>
<td>pipeline</td>
<td>coarse</td>
<td>medium</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>
ferret: similarity search

• Two serial stages and 4 middle parallel stages
  – Read a set of key images
  – Return the list of the most similar images in the database
ferret performance

- Target machine: HP9000 Superdome SMP (picasso)
  - 64 dual-core Itanium2, 1.6GHz, 380GB RAM, 9MB L3
ferret performance

• Efficiency problems:
  – Stage load imbalance and input bottleneck
dedup: data compression

- Main differences from ferret
  - S2 generate more output items than there were input
  - Some items may skip stage S4
dedup performance

• On Superdome

![Graph showing dedup performance on Superdome](image)
dedup performance

Same problems: load imbalance and output bottleneck
Scalability issues and solutions

• Scalability gated by
  – Load imbalance for small number of threads
  – I/O bottleneck for larger number of threads

• Solutions
  – Load imbalance
    • Collapsing all the parallel stages into one (not general)
    • Oversubscription: semi-dynamic scheduling
    • Work stealing: dynamic scheduling
  – I/O bottleneck
    • Parallel I/O
TBB vs. Pthreads

Improvement of 6-st. TBB over 6-st. Pthreads (ferret)

Improvement of 3-st. TBB over 3-st. Pthreads (ferret)

Improvement of 3-st. TBB over 3-st. Pthreads (dedup)
Summary of results (I)

• Advantages of using TBB to implement work stealing
  – Productivity (easier to write, # stages is not an issue)
  – Automatic load balance
  – Cache-conscious, thanks to the TBB pipeline template
  – Negligible overheads (measured with Vtune)

• PACT 2009:
Wavefront problems

• Wavefront is a programming pattern
  – Appears in important scientific applications
    • Dynamic programming or sequence alignment.
    • Checkerboard, Floyd, Financial, H.264 video compression.

• 2D wave problem

```
1  for (i=1; i<n; i++)
2       for (j=1; j<n; j++)
3           A[i,j] = foo(gs, A[i,j], A[i-1,j], A[i,j-1]);
```

• gs (grain size): vary the workload
2D Wavefront problem

A[i,j] = foo(gs, A[i,j], A[i-1,j], A[i, j-1]);
Task-based implementations (pseudocode)

Each element computation is done by a task

```
1 Task_Body(); //Task’s work
2 Critical Section {
3     counter[i+1][j]--; //Dec. south
4     if (counter[i+1][j]==0)
5         Spawn();
6 }
7 Critical Section {
8     counter[i][j+1]--; //Dec. east
9     if (counter[i][j+1]==0)
10    Spawn();
11 }
```
OpenMP 3.0

void Operation(int i, int j)
{
    int gs;
    bool ready;

    A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
    if (i<n-1) {
        omp_set_lock(&locks[i][j+1]);
        ready = --counter[i][j+1]==0;
        omp_unset_lock(&locks[i][j+1]);

        if (ready) {
            #pragma omp task
            Operation(i, j+1);}

        if (i<n-1){
            omp_set_lock(&locks[i+1][j]);
            ready = --counter[i][j+1]==0;
            omp_unset_lock(&locks[i+1][j]);

            if (ready) {
                #pragma omp task
                Operation(i+1, j);}
    }
}

- No atomic capture:
  - `--x==0` is not atomic
- omp critical & omp task

OpenMP_v1 (critical)
OpenMP_v2 (locks)

Memory consumption increases due to array of locks
TBB_v1 implementation

```cpp
1 Class Operation: public TBB::task
2 {
3     int i, j, gs;
4     public:
5         Operation(int i_, int j_) : i(i_), j(j_) {}  
6         task * execute();
7     };
8 TBB::task * Operation::execute()
9 {
10        A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
11        if (i<n-1) //There is south neighbor
12            if (--counter[i+1][j]==0)
13                spawn( ..... Operation(i+1,j) );
14        if (j<n-1) //There is east neighbor
15            if (--counter[i][j+1]==0)
16                spawn( ..... Operation(i,j+1) );
17    }
```

- TBB_v1 peculiarities
  - Atomic<int>
  - Override task::execute
  - spawn
TBB_v2 implementation

- TBB_v2 peculiarities
  - atomic <int>
  - parallel_do_feeder templ.
  - feeder.add method

```cpp
class MyBody{
public:
 MyBody() {};

void operator()(block& b, tbb::parallel_do_feeder<block> & feeder) const{

int i = b.first;
int j = b.second;
if (i<n && j<n){
    A[i][j] = foo(gs,A[i][j],A[i-1][j],A[i][j-1]);
    if (i<n-1 && --counter[i+1][j]==0)
        feeder.add(block(i+1,j));
    if (j<n-1 && --counter[i][j+1]==0)
        feeder.add(block(i,j+1));
}
}
```
Cilk Implementation

```c
void Operation(int i, int j)
{
    int gs;
    bool ready_e, ready_s;

    if (j<m-1) {
        pthread_mutex_lock(&locks[i][j+1]);
        counter[i][j+1]--;
        ready_e = (counter[i][j+1]==0);
        pthread_mutex_unlock(&locks[i][j+1]);
    }
    if (ready_e)
        cilk_spawn Operation(i, j+1);

    if (i<m-1) {
        pthread_mutex_lock(&locks[i+1][j]);
        counter[i+1][j]--;
        ready_s = (counter[i+1][j]==0);
        pthread_mutex_unlock(&locks[i+1][j]);
    }
    if (ready_s)
        cilk_spawn Operation(i, j+1);

    if (ready_e || ready_s)
        cilk_sync;
}
```

- Cilk peculiarities
  - No atomic operation
  - cilk_spawn, locks
  - cilk_sync
CnC Implemt.

```cpp
1 // Declarations. Tag collection to control execution
2 < par ElementTag >
3 // Step prescription: for each ElementTag instance
4 // we control an step exec.
5 <ElementTag>:: (Operation)
6 // Step execution: a step may produce a new ElementTag
7 (Compute) -> <ElementTag>
8 int Operation::execute(const par & t, wave & c ) const
9 {
  int i = t.first;
  int j = t.second;
  int gs;

  if (i < n-1)
    if (--counter[i+1][j] == 0)
      c.ElementTag.put(par(i+1, j));
  if (j < n-1)
    if (--counter[i][j+1] == 0)
      c.ElementTag.put(par(i, j+1));
  return CnC::CNC_Success;
}
```

- **CnC peculiarities**
  - Atomic<int>
  - Operation Collection
  - Element Tag

Sept 13, 2012

HPC-AC. Rafael Asenjo.
Experimental results

- Two Quad-Core Intel Xeon CPU X5355 at 2.66 GHz.
- Running example case study.
  - Matrix size = 1,000 x 1,000
- Speedup (with respect to the fastest sequential code)
- Task granularity:
  - Fine (200 FLOP), medium (2,000 FLOP) and coarse (20,000 FLOP) grain
Constant granularity: Speedups

Notes:
- Cilk sometimes does not complete
- Similar behavior of TBB_v1 and TBB_v2
- OpenMP has the worst scalability
- CnC is between TBB and OpenMP
Optimizations to reduce overheads in TBB

```c
1 Task_Body();    // Task’s work
2 if (j<n-1){    // There is east neighbor
3     if (--counters[i][j+1]==0)
4         recycle_into_east = true;
5 }
6 if (i<n-1){    // There is south neighbor
7     if (--counters[i+1][j]==0)
8         if (!recycle_into_east)
9             recycle_into_south = true;
10        else
11            spawn(i+1,j);
12 }
13 if (recycle_into_east){  //Recycle this into east
14     recycle_as_child_of();
15     j = j+1;
16     return this;
17 } else if(recycle_into_south){ //Recycle this into south
18     recycle_as_child_of();
19     i=i+1;
20     return this;
21 } else
22     return NULL;  // There is no neighbor task ready
```

Exploiting task passing mechanism available in TBB
→ TBB_v4

Number of spawns:
O(n²) → O(n)
Optimizations: reducing overheads in TBB

TBB_v3 is similar to TBB_v4 but without exploiting cache locality

- Scalability improvement: ~ 3x
- Overhead reduction: ~ 7x
- Cache improvement: ~ 4x
Wavefront template

```java
class Operation: public TBB::task
{
  int i, j;
  public:
    Operation(int i_, int j_): i(i_), j(j_)
    {
      task = execute();
    }

  TBB::task + Operation::execute()
  {
    // Task's work
    A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);

    if (j<n-1) // There is east neighbor
      if (--counters[i][j+1]==0)
        recycle_into_east = true;
    if (i<n-1) // There is south neighbor
      if (--counters[i+1][j]==0)
        if (recycle_into_east)
          recycle_into_south = true;
        else
          TBB_spawn(i+1, j);
    if (recycle_into_east)
      TBB_recycle_as_child_of();
      j = j+1;
      return this;
    }else if(recycle_into_south)
      TBB_recycle_as_child_of();
      i = i+1;
      return this;
    }else
      return NULL; // No ready neighbor

  int main()
  {
    atomic<int> *counters;
    ....
    for (i=1; i<n; i++)
      for (j=1; j<n; j++)
        if (i == 1) counters[i][j] = 1;
        else if (j == 1) counters[i][j] = 1;
        else counters[i][j] = 2;
    counters[1][1] = 0;
    TBB_task_scheduler_init init();
    TBB_spawn_root_and_wait (Operation(1,1));
    ....

    int i = GetFirst();
    int j = GetSecond();
    A[i][j]=foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
  }
}
```

Main file

```c
#include "wavefront.h"

void Operation::ExecuteTask()
{
  int i = GetFirst();
  int j = GetSecond();
  A[i][j]=foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
}

int main()
{
  ....
  wavefront_init(); // Initialize TBB and vars
  wavefront->run(); // execute the wavefront code
  ....
}
```

Definition file

```c
// Section 1: Data grid
[0:n-1, 0:n-1]

// Section 2: Task grid
[1:n-1, 1:n-1]

// Section 3: Indices
<i, j>

// Section 4: Dependency vectors
[1:n-2, 1:n-2] -> (0,1); (1,0)
[n-1, 1:n-2] -> (0,1)
[1:n-2, n-1] -> (1,0)

// Section 5 (Optional): counter values
[1,1] = 0
[1, 2:n-1] = 1
[2:n-1, 1] = 1
[2:n-1, 2:n-1] = 2
```

HPC-AC. Rafael Asenjo.
Wavefront template examples

a) Checkerboard

//Data grid
[0:m-1,0:n-1]
//Task grid
[1:m-1,::]
//Indices
(i,j)
//Dependence vectors
[1:m-1,1:n-2] -> (i-1,j); (i,0); (i,1)
[1:m-2,0] -> (1,j); (1,1)
[1:m-2,n-1] -> (i,1); (1,1)
//Counter values
[1,1] = 0
[2:m-1,0] = 2
[2:m-1,n-1] = 2
[2:m-1,1:n-2] = 3

b) Financial

//Data grid
[0:m-1,0:n-1]
//Task grid
[1:m-1,::]
//Indices
(i,j)
//Dependence vectors
[1:m-1,1:n-1] -> (i,0); (i,1)
[1:m-2,0] -> (1,j); (1,1)
[1:m-2,n-1] -> (i,1); (1,1)
//Counter values
[1,1:n-1] = 0
[2:m-1,0] = 2
[2:m-1,1:n-1] = j

HPC-AC. Rafael Asenjo.

Sept 13, 2012
Wavefront template results

1. Programmer specifies: i) definition file with dependence pattern information and ii) the task function.
2. The template is a productive tool (50% less programming effort) with low overhead costs (less than 5%).

Speedups of TBB manual vs. Template in real problems codes
Sumary of results (II)

• TBB features allow more efficient implementations:
  – Atomic capture and task recycling
• OpenMP greatly benefits from atomic capture
• CnC is competitive for coarse grain
• Cilk++ implementation is not robust enough
• HiPC 2011:
• ParCo 2011:
Tasks for heterogeneous

- Oversubscribe if there are GPUs
  - A dedicated thread wrapping the GPU
  - This thread has a work-stealing scheduler to feed the GPU
  - The GPU thread does not consume core cycles while the GPU is working
TBB for heterogeneous

- Higher level approach
  - TBB pipeline template:
    - A pipeline stage served by a GPU
      » Should be the bottleneck stage
  
- TBB Parallel_for template:
  - Adjust the grain size of the blocked_range iteration space for the tasks that will be mapped in the GPU
parallel_for with GPU

• Relying on a two-stages pipeline:
  – 1\textsuperscript{st} serial stage: generates chunks for CPU or GPU
  – 2\textsuperscript{nd} parallel stage: computes a chunk in CPU or GPU

• Second approach: variable chunk size
Experimental results

MxV: 8 cores

- CFS
- POX

MxV: 8 cores + 1 GPU

MxV: 8 cores + 2 GPUs

MxV: 8 cores + 4 GPUs

Sept 13, 2012

HPC-AC. Rafael Asenjo.
Experimental results

Barnes-Hut: 8 cores

Barnes-Hut: 8 cores + 1 GPU

Barnes-Hut: 8 cores + 2 GPUs

Barnes-Hut: 8 cores + 4 GPUs

Sept 13, 2012

HPC-AC. Rafael Asenjo.

Barnes-Hut: 8 cores

Barnes-Hut: 8 cores + 1 GPU

Barnes-Hut: 8 cores + 2 GPUs

Barnes-Hut: 8 cores + 4 GPUs

Sept 13, 2012

HPC-AC. Rafael Asenjo.
Experimental results

Ferret: 8 cores + 1 GPU

Ferret: 8 cores + 2 GPUs

Ferret: 8 cores + 4 GPUs
Conclusions

Advantages of task-based programming models

• Automatic load balancing
• Programmer can provide hints that improve cache behavior
• Portable; easier to parallelize the codes
• Well-suited for fine grain problems
• With some libraries:
  – Interoperability with other languages and tools
  – Templates available: parallel_for, pipeline, parallel containers…