Task inefficiency patterns for a wave equation solver

Gonzalo Gadeschi (NVIDIA), Oleksandr Rudyy (HLRS), Holger Schulz (Durham, UK), Tobias Weinzierl (Durham, UK)

IWOMP 2021

September 15, 2021
Executive summary and outline

▶ Application area: PDE solver ("Peano4/ExaHyPE2")
▶ Use task-based programming to solve loadbalancing problem
▶ Modelled with OpenMP5
▶ Tested runtimes don’t immediately allow us to do what we want
▶ Developed workarounds

1. Introduction
2. Observations
3. Solutions
Peano/Exahype

- Open source engine for simulation of wave phenomena
- Application agnostic design
- User input: system of PDEs
  - Tsunamis
  - Earthquakes
  - Gravitational waves
- Scalability: better resolution with bigger machines
Key components of the engine

- Adaptive mesh refinement with space filling curves
- Fixed and adaptive time stepping
- Mesh partitioning and load balancing
- Solver schemes (finite volumes, ADER-DG)
- There can be a lot of communication
Technologies

- Core code is written in C++
- Shared memory parallelism with OpenMP (cppthreads, tbb)
- Distributed memory parallelism with MPI
- GPU-offloading with OpenMP+CUDA
A simple BSP type solver

- Each thread processes a separate subpartition (taskloop).
- Can be very unbalanced
Test systems

- No MPI
- 3 different systems
- 3 different runtimes

| Test system | Hamilton                  | HPE Hawk                  | Cosma                      |
|-------------|---------------------------|---------------------------|----------------------------|
| CPU Name    | Intel Xeon E5-2650V4     | AMD EPYC 7742             | Intel Xeon Gold 5218       |
| Cores       | Broadwell 2 × 14          | Rome 2 × 64               | Cascade Lake 2 × 16        |
| NUMA domains| 2                         | 2 × 4                     | 2                          |
| Baseline freq.| 2.4 GHz                  | 2.25 GHz                  | 2.3 GHz                    |
| L2/L3        | 256 kB/30 MB              | 512 kB/16 MB              | 1 MB/22 MB                 |
| Compiler    | icpc (ICC) 19.1.3.304    | g++ (GCC) 10.2.0          | icpx (ICX) 2021.1 Beta     |
Scaling

- Well-balanced workload
- Not balanced workload

- Our BSP algorithm suffers badly from loadbalancing issues.
Enclave tasking

Identify all cells that share a boundary with another partition (as well as resolution changes)

These cells form the “skeleton” → critical path

All other cells form “enclaves” within their partition

They are *not* critical → asynchronous execution possible!
Enclave tasking vs BSP

- Loadbalancing problems can be severe
- Enclave tasking can mitigate problems
- Instead of a single sweep per time-step: two sweeps!
- Spawn lots of tasks in first sweep to smooth subsequent imbalances.
Enclave tasking algorithm

Algorithm 1 Schematic layout of the time-stepping in our enclave tasking.

```plaintext
function TIMES STEP(dt)
  #pragma omp taskloop nogroup
  for rank-local partition do
    for local cell do
      if cell is skeleton then
        update cell
      else
        #pragma omp task
        update cell
      end if
    end for
  end for
  #pragma omp taskwait
  Realise domain boundary exchange
  #pragma omp taskloop nogroup
  for rank-local partition do
    for local cell do
      if cell is enclave then
        busy-wait for enclave task outcome
      end if
    end for
  end for
  #pragma omp taskwait
  Implicitly wait for all tasks
end function
```

▶ **nogroup**: we don’t want to sync with the enclave tasks
Task graphs from OMPT

https://github.com/adamtuft/otter

BSP

Enclave tasking
Performance

As expected, no gain when work is nicely balanced.

But: Enclave tasking superior if there are imbalances!
Enclave tasking, how well does it work?

- Vtune, 4 threads, well-balanced setup:

- Vtune, 4 threads, unbalanced setup:

- We still have a lot of spinning.
- The situation is worse for unbalanced setups.
Zoom into single timestep (native OpenMP)

- Native OpenMP runtime, 4 threads
- Shaded area: 1 timestep in solver:
  - Reddish shade is first sweep (enclave task spawning)
  - Blueish shade is second sweep
- Queue threshold prevents spawning tasks, leaves very little ready tasks for second sweep, first sweep longer than second!
- Exactly the opposite of what we want!
Workaround

- OpenMP runtime eagerly processes tasks after some threshold reached
- `taskwait` allows processing *any* ready tasks $\rightarrow$ suspend task producer thread
- Almost all spawned tasks are consumed in first traversal

- New strategy “hold-back”:
  - Manual queue on top of OpenMP
  - No immediate task processing!
  - Busy polling processes task from queue $\rightarrow$ lazy task evaluation
Hold-back

- Much better: First sweep really short, large number of ready tasks spawned
- Somewhat faster, maybe 5%
Backfill

- Native: good for high throughput, discovers tasks that spawn more tasks early
- Hold-back: reduces latency along critical path

- New strategy “backfill” (on top of hold-back)
  - If number of *producer* tasks < number of threads → switch to enclave task processing (from manual queue)
  - Effectively work stealing
  - Compromise between throughput and latency
Both workarounds have desired effect.

Clear winner for our code with 4 threads is “backfill“ strategy — does it scale?
Scaling

As expected, no benefit in well-balanced setup

“backfill” keeps threads busy, robustly outperforms native OpenMP — as long as we utilise only one socket (14 cores Hamilton, 16 cores Cosma, 64 cores HAWK)
We use OpenMP5 tasks to solve a loadbalancing problem ("enclave tasking")

Tested 3 runtimes on 3 machines

Consistent observation: the runtimes don’t immediately support our ideas

We would like to:
- inform the runtime that we are about to spawn a ton of tasks that are not to be processed immediately
- annotate e.g. taskwait if throughput or latency take priority

Outlook: in upcoming work, investigate AMD machines closer (Archer2)