



A study of a composable approach to parallel programming for many-core multiprocessors

Vladimir Bakhtin, Nikita Kataev, Alexander Kolganov, Dmitry Zakharov,
Alexander Smirnov, Mikhail Kocharmin, and Anton Malakhov

Keldysh Institute of Applied Mathematics RAS

Rising Problem

Hardware

- A single-core performance improvement per year has dropped significantly.
- The road to higher performance is via the growth of the number of processor cores and the number of hardware threads per core.

Software

- The number of threads participating in the work increases.
- Each thread tends to execute smaller work portion.
- The thread arbitration and synchronization overheads rise.
- Available parallelism is limited by the Amdahl's law.

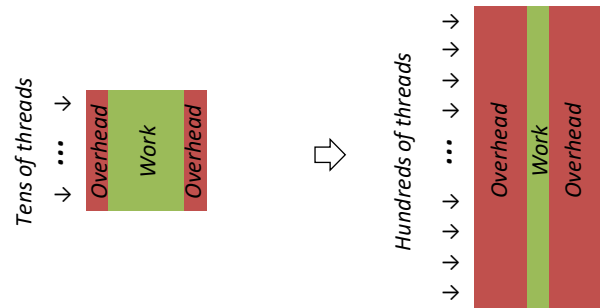
! **As a result, the gap between the parallel application performance and the peak performance of multiprocessors is getting wider.**

✓ **More parallelism is required!**

Possible sources of additional parallelism:

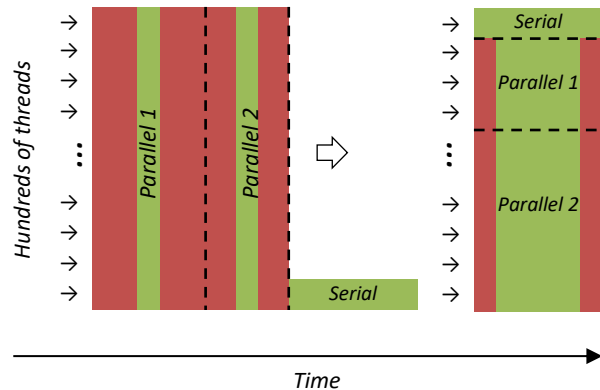
- Exploitation of nested parallelism.
- Execution of multiple different applications in parallel.

! **However, the direct use of OS-level threads to execute abundant parallelism often results in severe performance degradation by oversubscription of threads.**



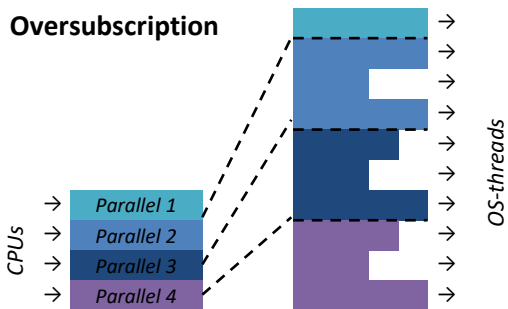
Kunpeng 920 (up to 192 threads)
Intel (up to 288 threads)
AMD (up to 768 threads)

Typical Frequency ~3.0 GHz



Composability Problem

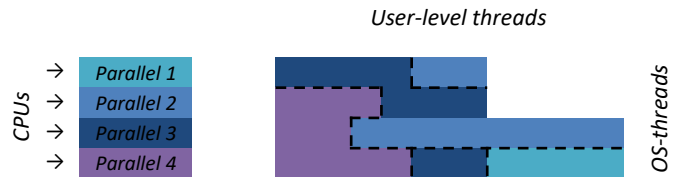
Oversubscription



Two levels of parallelism lead to $P \cdot P$ OS-level threads in the worst case (P – the number of CPUs).



Fine-grain space-time composition



We explore parallel programming patterns that pose significant problems for an application programmer and runtime developer to transparently replace OS-level threads with user-level threading and tasking abstractions.

We examine the performance different parallel programming models provide when facing oversubscription and other composability issues.

Different flexible models exist that implement lightweight threading and tasking runtimes:

- **oneAPI Threading Building Blocks (TBB)** – a C++ library that hides explicit OS-level threads under the higher level interface based on explicit tasks, however it suffers from inability to directly control program execution at low-level.
- **Argobots** - lightweight, low-level threading and tasking framework which proposes a rich set of controls, but requires a deep knowledge to achieve the best parallel program performance,
- **explicit OpenMP tasks (task and taskloop constructs)** – it is not flexible enough to provide various task orchestration strategies as TBB and Argobots do.

Different ways to implement user-level threads and fine-grain time-space composition.

The Prototype of the Compiler Infrastructure

Converter is a Clang-based source-to-source translator that generates parallel kernels and replaces OpenMP directives with runtime library function calls.

Supported OpenMP constructs:

- `parallel`, `parallel for` and `critical` directives,
- `reduction`, `private` and `num_threads` clauses,
- `omp_set_num_threads()`, `omp_get_max_threads()`, `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_num_procs()`, `omp_in_parallel()` functions.

A driver that simplifies an original program compilation and execution according to a chosen underlying programming model.

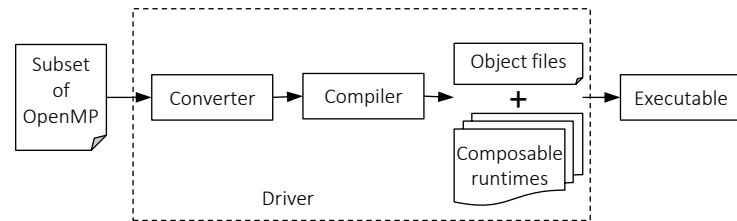
- Converter does not depend on the target runtime library
- Driver options allow us to choose a target runtime library at a link time.

Compiler is a normal compiler: GCC, Clang, etc.

We implemented 4 runtime libraries based on *TBB*, *Argobots*, normal OpenMP with explicit tasks (*task* and *taskloop* constructs).

The driver use environment variables to manage program execution:

- `CP_NUM_STREAMS`, `CP_NUM_THREADS_HINT`, `CP_STACK_SIZE`, `CP_PARTITIONER_KIND`,
`CP_NUM_NUMAS`, `CP_NUMAS`, `CP_STREAM_BIND`, `CP_THREAD_BIND`, etc.



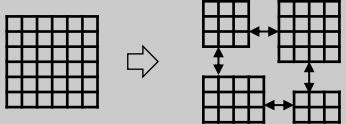
➤ `./cpt c -tbb file.c -o file.tbb.out`

➤ `CP_NUM_STREAMS=28`
`CP_NUM_THREADS_HINT=10000`
`./cpt run file.tbb.out`



The NAS Parallel Benchmarks 3.3

NPB are widely used in many research activities for evaluating specific architectures and systems and to evaluate automatic and semi-automatic parallelization techniques.

	Application	Description	Features	
Basic kernels	EP (Embarassingly Parallel)	generates a large number of independent Gaussian random deviates using the Marsaglia polar method	highly intensive computations on independent data	One level of parallelism
	MG (Multi Grid)	approximates the solution for a three-dimensional discrete Poisson equation using the V-cycle multigrid method	memory bound, conjunction of short- and long-distance communication patterns	
	FT (Fast Fourier Transform)	solves a three-dimensional partial differential equation (PDE) using the fast Fourier transform (FFT)	long-distance memory accesses	
	CG (Conjugate Gradient)	approximates the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration method together with the conjugate gradient method	a lot of irregular accesses, a lot of floating point reduction operations at each iteration	
	IS (Integer Sort)	performs an integer sorting among a sparse set of numbers. By default, it implements the Bucket-Sorting algorithm	memory bound, integer computations	
Simulated application programs	BT (Block Tridiagonal)	solve a synthetic system of nonlinear PDEs (three-dimensional system of Navier-Stokes equations for compressible fluid or gas) using three different iterative methods	different amount of parallel computations that are limited with loop-carried data dependencies	
	SP (Scalar Pentadiagonal)			
	LU (Lower-Upper)			
Multi zone	BT-MZ	re-implements three simulation application programs in a way that in each program a three-dimensional mesh is divided into two-dimensional horizontal tiling of three-dimensional zones		Two levels of parallelism
	SP-MZ			
	LU-MZ			



Transformation of the NAS NPB

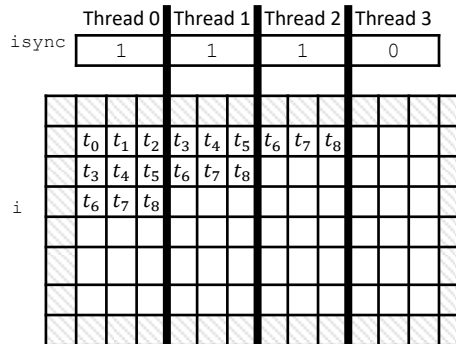
1. We examined C++ versions (original OpenMP versions) of benchmarks obtained from the original Fortran versions.
2. We moved each parallel pattern in a separate parallel region to satisfy our compiler requirements and to provide a runtime with more ways to schedule execution of different parallel patterns (composable OpenMP versions).
3. We encapsulate all data and functions accessed in each benchmark into a C++ class with a single public function that executes the benchmark. It allows us to execute multiple benchmarks from a single program in a composable way.
4. We remove load balancing features implemented in NPB-MZ benchmarks to estimate the capability of different runtimes to perform load balancing implicitly.
5. We replace pipelining approach in LU benchmark with hyperplane algorithm to remove extra synchronization between parallel threads, while preserving loop-carried data dependencies over all three array dimension.
6. The IS benchmark imposes restrictions on possible work distribution between threads from different parallel regions. We perform the scheduling manually and determine the relation between the thread index and the indexes of keys to sort.

```
#pragma omp parallel
{
  #pragma omp for
  ...
}
```



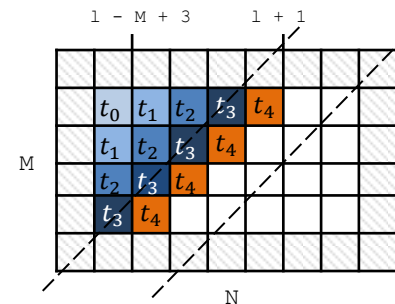
```
#pragma omp parallel for
{
  ...
}
```

(2)



Pipeline, original implementation, 2D example

```
for (int l = 1; l < M + N - 4; ++l) {
  #pragma omp parallel for
  for (int j = max(1, l - M + 3); j < min(l + 1, N - 1); ++j) {
    int i = l - j + 1;
    a[i][j] = (a[i - 1][j] + a[i + 1][j] + a[i][j - 1] + a[i][j + 1]) / 4;
  }
}
```



Hyperplane, 2D example

(5)

```
#pragma omp parallel
{
  #pragma omp for schedule (static)
  ...
}
#pragma omp parallel
{
  #pragma omp for schedule (static)
  ...
}
```



```
#pragma omp parallel
{
  // Calculate loop subrange based
  // on thread id and number of threads
}
#pragma omp parallel
{
  // Calculate loop subrange based
  // on thread id and number of threads
}
```

(6)



Experiments

2 multicore platforms

- Xeon – 2 x Intel Xeon CPU E5-2680 v4 @ 2.4 GHz (x86 architecture, 2 OS-level threads per core, 56 OS-level threads in total), 64 GB of main memory, Ubuntu 22.04.4 LTS.
- Kunpeng – 2 x Kunpeng 920 CPU @ 2600 GHz (aarch64 architecture, 1 OS-level thread per core, 48 OS-level threads in total) with 512 GB of main memory, CentOS Linux 8.

Compilation and execution environment

- Miniconda package manager, TBB 2021.11, GCC 13.2.0 with `-O3` option.

Problem sizes

- Mostly class A to simulate the decrease of the work portion per thread.
- Class C was also used for some experiments.

Correctness check

- Built-in verification functions implemented as parts of NPB and NPB-MZ.

Time measurement

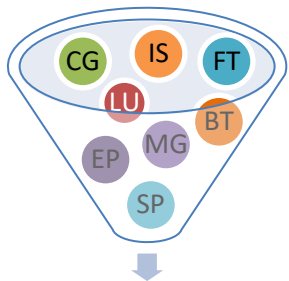
- NPB-MZ – built-in functions that enclose only computational parts of the benchmarks and exclude data preparation and data verification steps from the evaluation.
- NPB – the overall execution time is measured because we execute multiple benchmarks simultaneously to explore the performance of the composable execution.

Name	Parameter	Class A	Class C
CG	no. of rows	14000	150000
	no. of nonzeros	11	15
	no. of iterations	15	75
	eigenvalue shift	20	110
EP	no. of random pairs	2^{28}	2^{22}
FT	grid size	256x256x128	512x512x512
	no. of iterations	6	20
IS	no. of keys	2^{23}	2^{27}
	key max. value	2^{19}	2^{23}
MG	grid size	256x256x256	512x512x512
	no. of iterations	4	20
BT	grid size	64x64x64	162x162x162
	no. of iterations	200	200
	time step	0.0008	0.0001
LU	grid size	64x64x64	162x162x162
	no. of iterations	250	250
	time step	2.0	2.0
SP	grid size	64x64x64	162x162x162
	no. of iterations	400	400
	time step	0.0015	0.00067

Class	BT-MZ		LU-MZ		SP-MZ		Aggregated grid Size
	no.zones	no.iters	no.zones	no.iters	no.zones	no.iters	
A	4x4	200	4x4	250	4x4	400	128x128x16
B	16x16	200	4x4	250	16x16	400	480x320x28

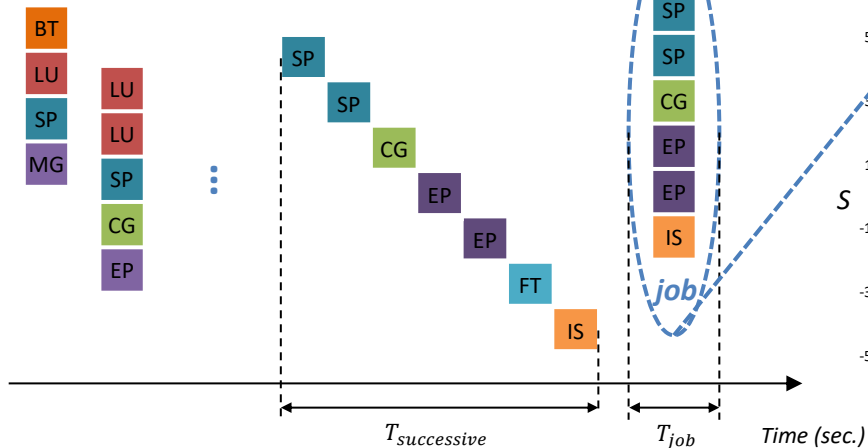


Composable Execution of NPB



$$T_{\text{successive}} = \sum_{b \in \text{benchmarks}} T_b$$

$$S = \frac{T_{\text{successive}} - T_{\text{job}}}{T_{\text{successive}}} * 100\%$$

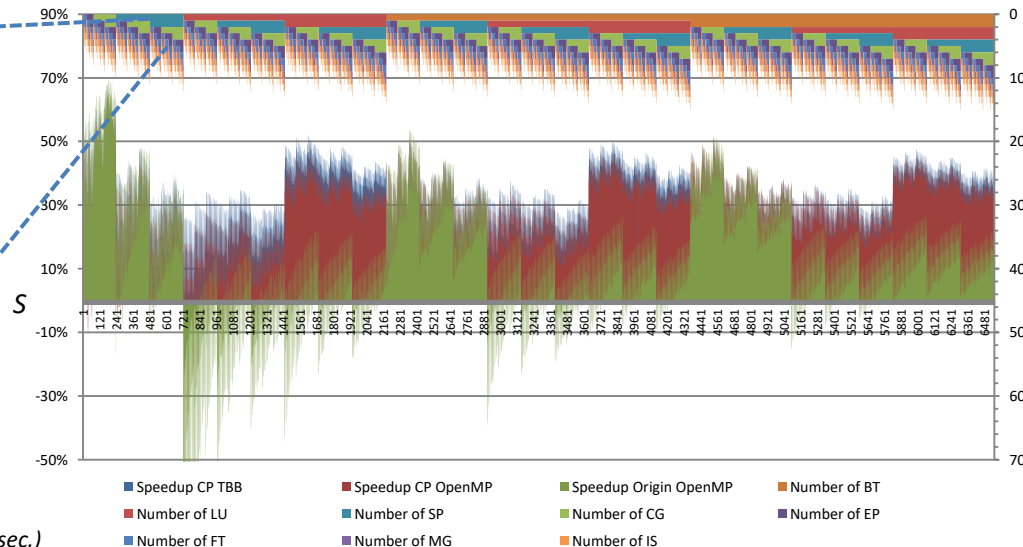


To calculate $T_{\text{successive}}$ we have used three versions of benchmarks: original OpenMP, composable OpenMP, one of our composable runtime.

```
// Successive execution of all jobs
for (auto &job : list_of_jobs) {
    // Parallel execution of all benchmarks in a job
    // - first level of parallelism.
    auto t_start = time();
    #pragma omp parallel for
    for (auto &benchmark : job) {
        // Parallel execution of each benchmark
        // - second level of parallelism
        benchmark.execute();
    }
    auto t_job = time() - t_start();
}
```

.VS

```
for (auto &benchmark : job) {
    auto t_start = time();
    benchmark.execute();
    auto t_b = time() - t_start();
}
```

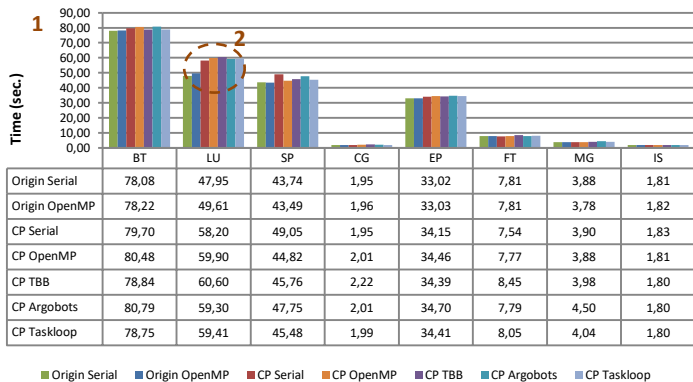


Class A, TBB based runtime, Kunpeng
48 OS-level threads, auto user-level threads



Independent Evaluation of each Benchmark on Different Runtimes

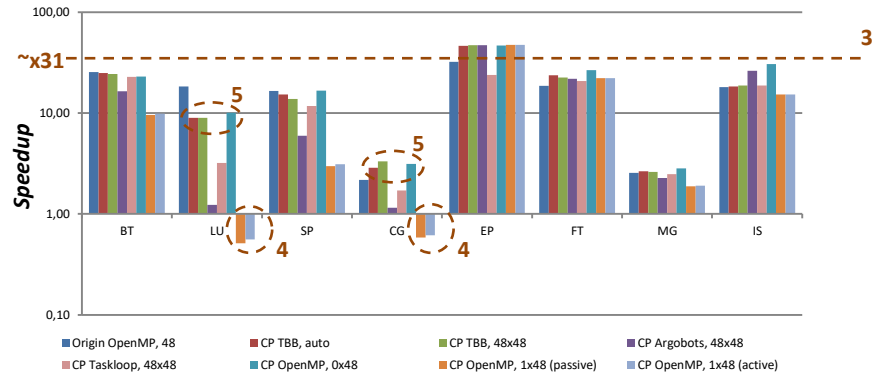
Kunpeng



1 OS-level thread

1. The measurements illustrate that a chosen runtime almost does not affect the execution time if only a single thread is used.
2. The use of hyperplane algorithm instead of pipelining in the composable version of LU makes this benchmark slowdown noticeably

* The CP abbreviation means the composable version of the benchmark. For speedup results the number of user-level threads at each level is shown for each runtime except benchmarks compiled with the natural OpenMP runtime. In that case the number of OS-level threads at each level is specified, and thread affinity is enabled and it is set to `OMP_PROC_BIND=spread,close`. We also manually set the `OMP_WAIT_POLICY` environment variable to `active` or `passive` values. If the number of threads is zero, a corresponding level of parallelism is disabled and associated `parallel` directive is removed from sources.



48 OS-level thread

3. Results for standalone execution of each benchmark highlights the underutilization issue because a speedup never reaches the available number of hardware threads.
4. The natural OpenMP runtime shows drastic degradation even if only one OS-level thread explicitly set to execute an outer level of parallelism.
5. The TBB based runtime provides almost the same performance as the native OpenMP while the Argobots library and the OpenMP taskloop construct degrade the performance in case of LU and CG benchmarks due to:
 - mutex-based implementation of reduction in Argobots and equal number of threads for different work sizes,
 - ineffective work arbitration and synchronization strategy implemented in OpenMP taskloop based runtime.



Composable Execution of NPB

The drop of the performance of some benchmarks in comparison with the performance of the original OpenMP versions impacts the speedup if the baseline is a successive execution of OpenMP versions. However, the TBB based runtime allows us to achieve up to the 70% speedup improvement even in that case.

```
// Successive execution of all jobs
for (auto &job : list_of_jobs) {
    // Parallel execution of all benchmarks in a job
    // - first level of parallelism.
    auto t_start = time();
    #pragma omp parallel for
    for (auto &benchmark : job) {
        // Parallel execution of each benchmark
        // - second level of parallelism
        benchmark.execute();
    }
    auto t_job = time() - t_start();
}
```

.VS

```
for (auto &benchmark : job) {
    auto t_start = time();
    benchmark.execute();
    auto t_b = time() - t_start();
}
```

$$T_{\text{successive}} = \sum_{b \in \text{benchmarks}} T_b$$

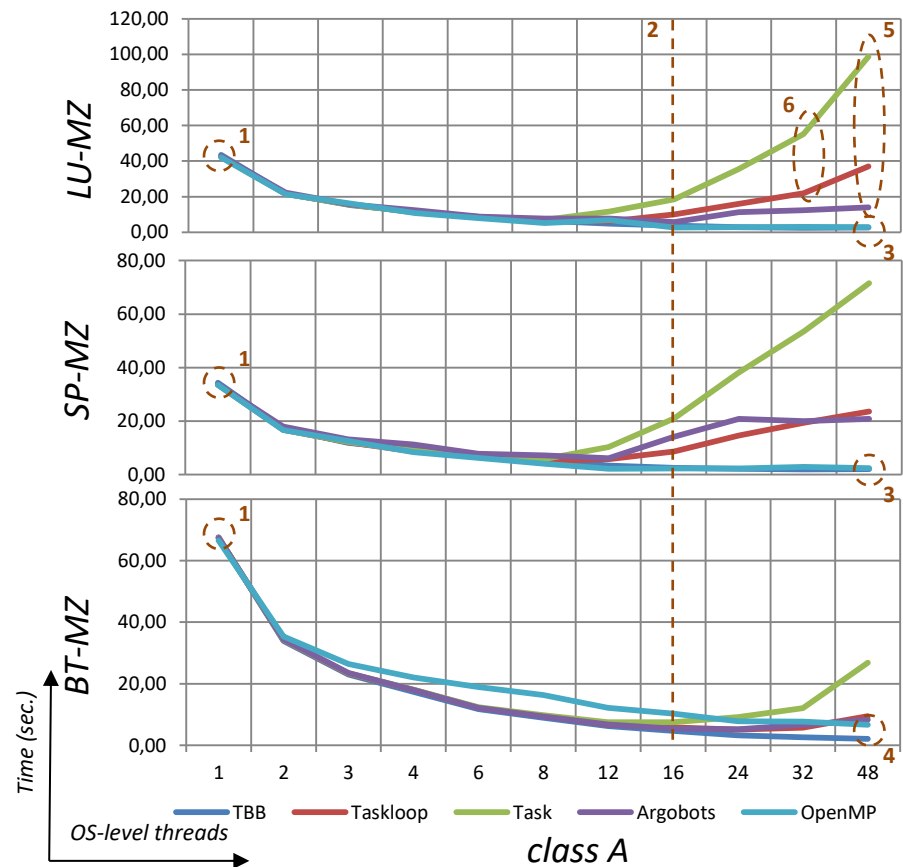
$$S = \frac{T_{\text{successive}} - T_{\text{job}}}{T_{\text{successive}}} * 100\%$$

Table. Comparing composable execution of NPB class A

Platform	Runtime	Threads		Jobs	Max. number of benchmarks in a job								Time (h.)	Speedup to successive execution								
		OS	User		BT	LU	SP	CG	EP	FT	MG	IS		Self			CP OpenMP			Origin OpenMP		
														Total	Min	Max	Total	Min	Max	Total	Min	Max
Kumpeng	TBB	48	Auto	6560	2	2	2	2	2	2	2	2	19.38	34%	-2%	65%	30%	-67%	62%	13%	-147%	70%
	TBB	48	48x48	6560	2	2	2	2	2	2	2	2	20.08	33%	-56%	63%	28%	-71%	61%	10%	-148%	69%
	Taskloop	48	48x48	6560	2	2	2	2	2	2	2	2	39.67	29%	-5%	72%	-43%	-217%	60%	-77%	-598%	68%
	Argobots	48	48x48	2186	2	2	2	2	2	2	2	2	14.2	59%	-36%	81%	-54%	-632%	58%	-92%	-1515%	67%
Xeon	TBB	56	56x56	6560	2	2	2	2	2	2	2	2	42.83	8%	-51%	41%	-15%	-121%	37%	-37%	-121%	40%
	Taskloop	56	56x56	6560	2	2	2	2	2	2	2	2	41.46	4%	-16%	49%	-12%	37%	-113%	-33%	40%	-115%



Execution Time of Composable Versions of NPB-MZ on Kunpeng

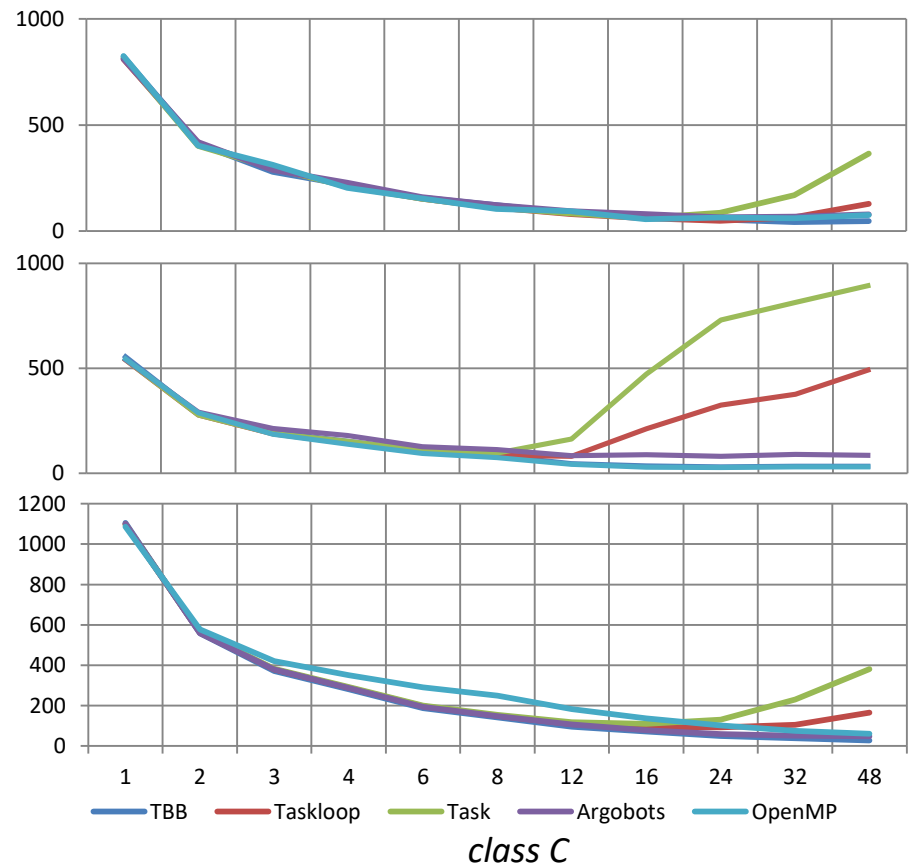
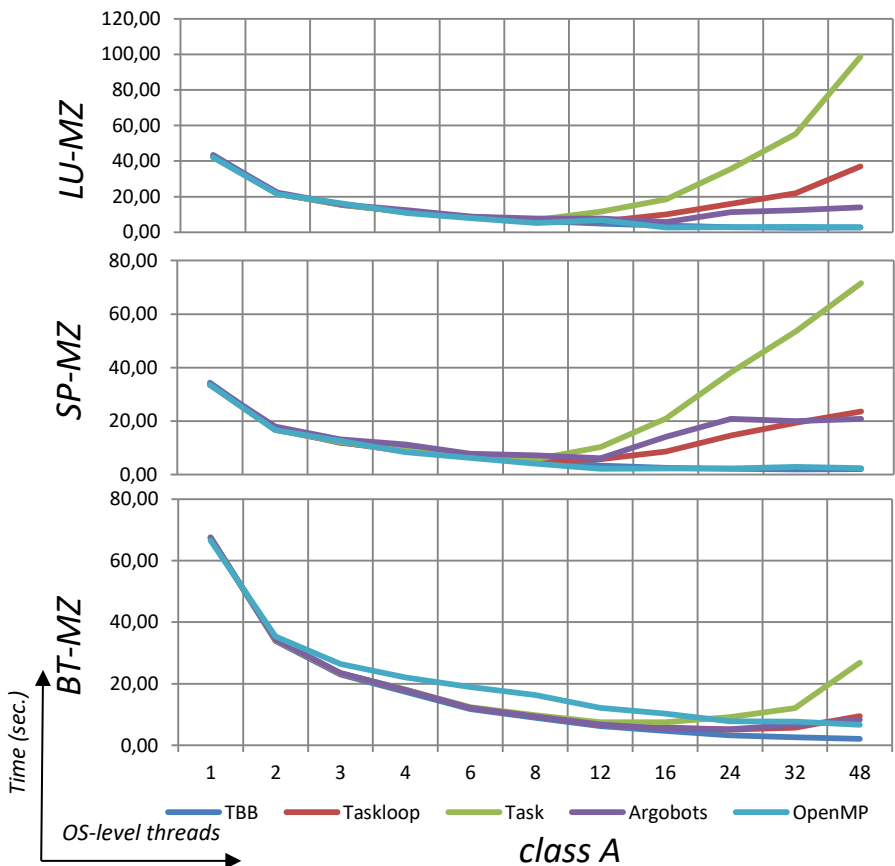


In case of the native OpenMP runtime an optimal number of OS-level threads are manually selected at each level of parallelism.

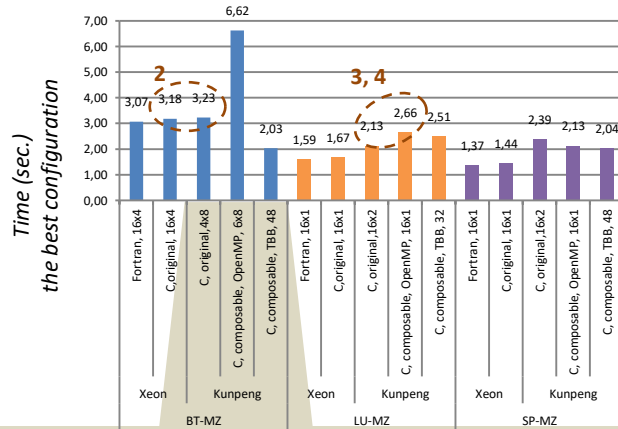
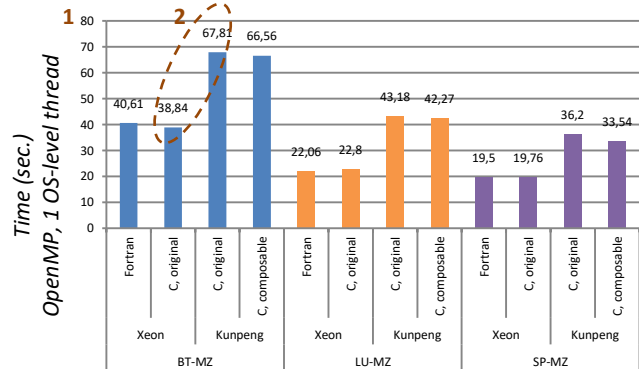
1. In case of a single thread the execution time is very close and neither programming model gains an advantage.
2. The LU-MZ and SP-MZ creates 16 zones of equal sizes and our experiments shows that an outer level of parallelism is enough to achieve the good performance and the benchmark versions based on the native OpenMP runtime stop scaling around 16 OS-level threads.
3. For LU-MZ and SP-MZ the TBB based implementation of the nested parallelism allows us to achieve slightly better performance if more than 16 OS-level threads are used.
4. The measurements indicate that in case of BT-MZ the TBB based runtime allows as to achieve twice the performance of the OpenMP version on 48 OS-level threads.
5. The better scaling in case of Argobots based runtime is the result of lightweight structure of Argobots user-level threads in comparison to OpenMP explicit tasks.
6. In case of runtime based on OpenMP task constructs a single thread in a team create each task in a sequential loop. Thus this runtime stop scaling with the least number of threads the runtime based on OpenMP taskloop constructs.



Execution Time of Composable Versions of NPB-MZ on Kunpeng



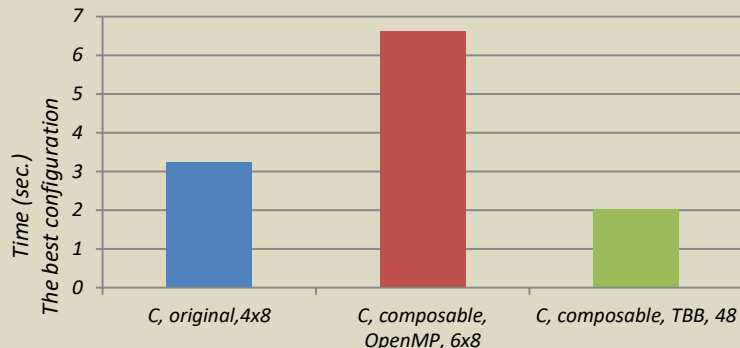
Comparing the Original and Composable Versions of NPB-MZ (class A)



The optimal number of OS-level threads at each level of parallelism has been found in a manual way.

1. The execution time of different versions obtained using OpenMP running 1 OS-level thread are very close.
2. The C versions directly translated from the Fortran versions scales better on the Kunpeng platform.
3. The composable versions compiled with the native OpenMP runtime provide lower performance due to each parallel loop is enclosed into a separate parallel region.
4. In LU-MZ we use a hyperplane algorithm to preserve the data dependencies instead of the pipelining strategy applied to the original benchmark. Thus, every time new hyperplane is processed a thread encounters a parallel construct.

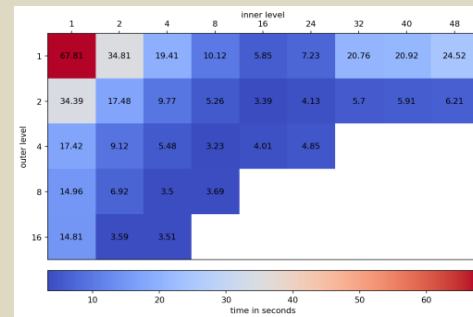
- Comparing the best performance of the original and composable versions of BT from NPB-MZ class A on Kunpeng 920.
- The different numbers of OS-level threads at each level of parallelism has been explored for OpenMP programs.
- Up to the maximum number of OS-level threads has been explored for TBB-based runtime.



Manual load balancing.
Manual number of threads tuning.

No load balancing.
Manual number of threads Automatic number of parallel workers tuning.

Automatic load balancing.
Automatic number of parallel workers tuning.



BT-MZ, class A, original. Execution time (sec.) using different number of OS-level threads at each level of parallelism.



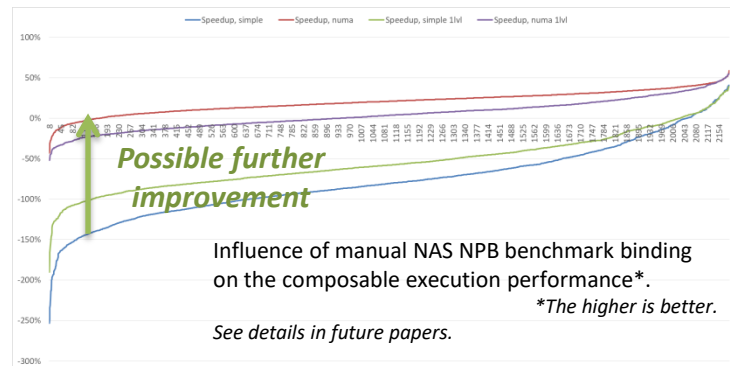
Conclusion

We are looking for the way to overcome scalability wall and we believe that our research may affect the way how future multithreaded libraries should be organized.

- Parallel programming models directly build on top of OS-level threads is inadequate in the case of oversubscription and should be transparently replaced with lightweight approaches that rely on user-level threads.
- The implementation of runtimes of well-known models, such as OpenMP, can be changed to efficiently handle composable parallelism. However, it still may require an application programmer to update their sources to achieve better performance.

We can increase the throughput by an average of 40% using existing hardware and software (with little changes).

Future work should be done to better estimate the influence of NUMA architecture, the task binding and the better number of threads per task on the application performance.



Thank you for your attention



URL

<http://dvm-system.org>

dvm@keldysh.ru



E-mail

К
Г
М
RAS
DvM
SYSTEM



Russian Supercomputing Days 2024