electronics

MDPI

Article

Assessment of OpenMP Master–Slave Implementations for Selected Irregular Parallel Applications

Paweł Czarnul

Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland; pczarnul@eti.pg.edu.pl; Tel.: +48-58-347-12-88

Abstract: The paper investigates various implementations of a master–slave paradigm using the popular OpenMP API and relative performance of the former using modern multi-core workstation CPUs. It is assumed that a master partitions available input into a batch of predefined number of data chunks which are then processed in parallel by a set of slaves and the procedure is repeated until all input data has been processed. The paper experimentally assesses performance of six implementations using OpenMP locks, the tasking construct, dynamically partitioned for loop, without and with overlapping merging results and data generation, using the gcc compiler. Two distinct parallel applications are tested, each using the six aforementioned implementations, on two systems representing desktop and worstation environments: one with Intel i7-7700 3.60 GHz Kaby Lake CPU and eight logical processors and the other with two Intel Xeon E5-2620 v4 2.10 GHz Broadwell CPUs and 32 logical processors. From the application point of view, irregular adaptive quadrature numerical integration, as well as finding a region of interest within an irregular image is tested. Various compute intensities are investigated through setting various computing accuracy per subrange and number of image passes, respectively. Results allow programmers to assess which solution and configuration settings such as the numbers of threads and thread affinities shall be preferred.

Keywords: master–slave; parallel programming; OpenMP; thread affinity

check for updates

Citation: Czarnul, P. Assessment of OpenMP Master–Slave Implementations for Selected Irregular Parallel Applications. *Electronics* **2021**, *10*, 1188. https://doi.org/10.3390/electronics10101188

Academic Editor: Antonio F. Díaz

Received: 16 April 2021
Accepted: 12 May 2021
Published: 16 May 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In today's parallel programming, a variety of general purpose Application Programming Interfaces (APIs) are widely used, such as OpenMP, OpenCL for shared memory systems including CPUs and GPUs, CUDA, OpenCL, OpenACC for GPUs, MPI for cluster systems or combinations of these APIs such as: MPI+OpenMP+CUDA, MPI+OpenCL [1], etc. On the other hand, these APIs allow to implement a variety of parallel applications falling into the following main paradigms: master–slave, geometric single program multiple data, pipelining, divide-and-conquer.

At the same time, multi-core CPUs have become widespread and present in all computer systems, both desktop and server type systems. For this reason, optimization of implementations of such paradigms on such hardware is of key importance nowadays, especially as such implementations can serve as templates for coding specific domain applications. Consequently, within this paper we investigate various implementations of one of such popular programming patterns—master–slave, implemented with one of the leading APIs for programming parallel applications for shared memory systems—OpenMP [2].

2. Related Work

Works related to the research addressed in this paper can be associated with one of the following areas, described in more detail in subsequent subsections:

1. frameworks related to or using OpenMP that target programming abstractions even easier to use or at a higher level than OpenMP itself;

2. parallelization of master–slave and producer–consumer in OpenMP including details of analyzed models and proposed implementations.

2.1. OpenMP Related Frameworks and Layers for Parallelization

SkePU is a C++ framework targeted at using heterogeneous systems with multi-core CPUs and accelerators. In terms of processing structures, SkePU incorporates several skeletons such as Map, Reduce, MapReduce, MapOverlap, Scan and Call. SkePU supports back-ends such as: sequential CPU, OpenMP, CUDA and OpenCL. Work [3] describes a back-end for SkePU version 2 [4] that allows to schedule workload on CPU+GPU systems. Better performance than the previous SkePU implementation is demonstrated. The original version of hybrid execution in SkePU 1 ran StarPU [5] as a back-end. The latter allows to encompass input data with codelets that can be programmed with C/C++, OpenCL and CUDA. Eager, priority and random along with caching policies are available for scheduling. Workload can be partitioned among CPUs and accelerators either automatically or manually with a given ratio. Details of how particular skeletons are assigned in shown in [3]. In terms of modeling, autotuning is possible using a linear performance formulation and its speed-ups were shown versus CPU and accelerator implementations.

OpenStream is an extension of OpenMP [6,7] enabling to express highly dynamic control and data flows between dependent and nested tasks. Additional `input` and `output` clauses for the `task` construct are proposed with many examples presenting expressiveness of the approach. There is a trade-off between expressiveness of streaming annotations and overhead which is studied versus Cilk using an example of recursive Fibonacci implementations and obtaining granularity threshold above which task's overheads are amortized. Additional clauses `peek` and `tick` allow reading data from a stream without advancing the stream as well as advancing the read index in streams. Speed-ups of the proposed OpenMP streaming solution over sequential LAPACK are demonstrated [6] for Cholesky factorization of various sizes (up to superlinear 27.4 for 4096 $\times$ 4096 matrix size and 256 numbers of blocks per matrix, due to caching effects) as well as SparseLU (19.5 for block size 64 $\times$ 64 and 22 for block size 128 $\times$ 128 for 4096+ numbers of blocks). A Gauss–Seidel algorithm using the proposed solution, hand coded, achieved the speed-ups of 8.7 for 8k $\times$ 8k matrix and 256 $\times$ 256 tile size. Tests were conducted on a dual-socket AMD Opteron Magny-Cours 6164HE machine with 2 $\times$ 12 cores running at 1.7 GHz and 16 GB of RAM.

Argobots [8] is a lightweight threading layer that can be used for efficient processing and coupling high-level programming abstractions to low-level implementations. Specifically, the paper shows that the solution outperforms state-of-the-art generic lightweight threading libraries such as MassiveThreads and Qthreads. Additionally, integrating Argobots with MPI and OpenMP is presented with better performance of the latter for an application with nested parallelism than competing solutions. For the former configuration, better opportunities regarding reduction of synchronization and latency are shown for Argobots compared to Pthreads. Similarly, better performance of Argobots vs Pthreads is discussed for I/O operations.

PSkel [9], as a framework, targets parallelization of stencil computations, in a CPU+GPU environment. As its name suggests, it uses parallel skeletons and can use NVIDIA CUDA, Intel TBB and OpenMP as back-ends. Authors presented speed-ups of the hybrid CPU+GPU version up to up to 76% and 28%, versus CPU-only and GPU-only codes.

Paper [10] deals with introduction of another source layer between a program with OpenMP constructs and actual compilation. This approach translates OpenMP constructs into an intermediate layer (NthLib is used) and the authors are advocating future flexibility and ease of introduction of changes into the implementation in the intermediate layer.

Optimization of OpenMP code can be performed at a lower level of abstraction, even targeting specific constructs. For example, in paper [11] authors present a way for automatic transformation of code for optimization of arbitrarily-nested loop sequences with affine dependencies. An Integer Linear Programming (ILP) formulation is used for finding good tiling hyperplanes. The goal is optimization of locality and communication-minimized

MOST WIEDZY Downloaded from mostwiedzy.pl

coarse-grained parallelization. Authors presented notable speed-ups over state-of-the-art research and native compilers ranging from 1.6x up to over 10x, depending on the version and benchmark. Altogether, significant gains were shown for all of the following codes: 1-d Jacobi, 2-d FDTD, 3-d Gauss-Seidel, LU decomposition and Matrix Vec Transpose.

In paper [12], the author presented a framework for automatic parallelization of divide-and-conquer processing with OpenMP. The programmer needs to provide code for key functions associated with the paradigm, i.e., data partitioning, computations and result integration. Actual mapping of computations onto threads is handled by the underlying runtime layer. The paper presents performance results for an application of parallel adaptive quadrature integration with an irregular and imbalanced corresponding processing tree. Obtained speed-ups using an Intel Xeon Phi system reach around 90 for parallelization of an irregular adaptive integration code which was compared to a benchmark without thread management at various levels of the divide-and-conquer tree which resulted in maximum speed-ups of 98.

OmpSs [13] (https://pm.bsc.es/ompss (accessed on 16 April 2021)) extends OpenMP with new directives to support asynchronous parallelism and heterogeneity (like GPUs, FPGAs). A target construct is available for heterogeneity implementation, data dependencies can be defined between various tasks of the program. Functions can also be annotated with the task construct. While OpenMP and OmpSs are similar, some differences exist (https://pm.bsc.es/ftp/ompss/doc/user-guide/faq-openmp-vs-ompss.html (accessed on 16 April 2021)). Specifically, in OmpSs, that uses `#pragma omp` directives, one creates work using `#pragma omp task` or `#pragma omp for` and the program already starts with a team of threads out of which one executes the main function. `#pragma omp parallel` is ignored. For the `for` loop, a compiler creates a task which will create internally several more tasks out of which each implements some part of the iteration space of the corresponding parallel loop. OmpSs has been an OpenMP forerunner for some of the features [14,15]. Recent paper [16] presents an architecture and a solution that extends the OmpSs@FPGA environment with the possibility for the tasks offloaded to FPGA to create and synchronize nested tasks without the need to involve the host. OmpSs-2, following its specification (https://pm.bsc.es/ftp/ompss-2/doc/spec (accessed on 16 April 2021)), extends the tasking model of OmpSs/OpenMP so that both task nesting and fine-grained dependencies across different nesting levels are supported. It uses `#pragma oss` constructs. Important features include, in particular: nested dependency domain connection, early release of dependencies, weak dependencies, native offload API task Pause/Resume API. It should be noted that the latest OpenMP standard also allows tasking as well as offloading to external devices such as Intel Xeon Phi or GPUs [2].

Paper [17] presents PLASMA—the Parallel Linear Algebra Software for Multicore Architectures—a version which is an OpenMP task based implementation adopting a tile-based approach to storage, along with algorithms that operate on tiles and use OpenMP for dynamic scheduling based on tasks with dependencies and priorities. Detailed assessment of the software performance is presented in the paper using three platforms with 2 × Intel Xeon CPU E5-2650 v3 CPUs at 2.3 GHz, Intel Xeon Phi 7250 and 2 × IBM POWER8 CPUs at 3.5 GHz, respectively, using gcc compared to MKL (for Intel) and ESSL (for IBM). PLASMA resulted in better performance for algorithms suited for its tile type approach such as LDL^T factorization as well as QR factorization in the case of tall and skinny matrices.

In [18] authors presented parts of the first prototype of sLaSs library with auto tunable implementations of operations for linear algebra. They used OmpSs with its task based programming model and features such as weak dependencies and regions with the final clause. They benchmarked their solution using a supercomputer featuring nodes with 2 sockets with Intel Xeon Platinum 8160 CPUs, with 24 cores and 48 logical processors. Results are shown for TRSM for th original LASs, sLaSs and PLASMA, MKL and ATLAS, for NPGETRF for LASs, sLaSs and MKL and for NPGESV for LASs and sLaSs demonstrating improvement of the proposed solution of about 18% compared to LASs.

MOST WIEDZY Downloaded from mostwiedzy.pl

2.2. Parallelization of Master–Slave with OpenMP

master–slave can be thought of as a paradigm to enable parallelization of processing among independently working slaves that receive input data chunks from the master and return results to the master.

OpenMP by itself offers ways of implementing the master–slave paradigm, in particular using:

1. `#pragma omp parallel` along with `#pragma omp master` directives or `#pragma omp parallel` with distinguishing master and slave codes based on thread ids.
2. `#pragma omp parallel` with threads fetching tasks in a critical section, a counter can be used to iterate over available tasks. In [19], it is called an all slave model.
3. Tasking with the `#pragma omp task` directive.
4. Assignment of work through dynamic scheduling of independent iterations of a `for` loop.

In [19], the author presented virtually identical and almost perfectly linear speed-up of the all slave model and the (dynamic,1) loop distribution for the Mandelbrot application on 8 processors. In our case, we provide extended analysis of more implementations and many more CPU cores.

In work [20], authors proposed a way to extend OpenMP for master–slave programs that can be executed on top of a cluster of multiprocessors. A source-to-source translator translates programs that use an extended version of OpenMP into versions with calls to their runtime library. OpenMP's API is proposed to be extended with `#pragma domp parallel taskq` for initialization of a work queue and `#pragma domp task` for starting tasks as well as `#pragma domp function` for specification of MPI description for the arguments of a function. The authors presented performance results for applications such as computing Fibonacci numbers as well as embarrassingly parallel examples such as generation of Gaussian random deviates and Synthetic Matrix Addition showing very good scalability with configurations up to 4×2 and 8×1 (processes $\times$ threads). More interesting in the context of this paper were results for MAND which is a master–slave application that computes the Mandelbrot set for a 2-d image of size 512×512 pixels. Speed-up on an SMP machine for the best 1×4 configuration (4 CPUs) amounted to 3.72 while on a cluster of machines (8 CPUs) was 6.4, with a task stealing mechanism.

OpenMP will typically be used for parallelization within cluster nodes and integrated with MPI at a higher level for parallelization of master–slave computations among cluster nodes [1,21]. Such a technique should yield better performance in a cluster with multicore CPUs than an MPI only approach in which several processes are used as slaves as opposed to threads within a process communicating with MPI. Furthermore, overlapping communication and computations can be used for earlier sending out data packets by the master for hiding slave idle times. Such a hybrid MPI/OpenMP scheme has been further extended in terms of dynamic behavior and malleability (ability to adapt to a changing number of processors) in [22]. Specifically, the authors have implemented a solution and investigated MPI's support in terms of needed features for an extended and dynamic master/slave scheme. A specific implementation was used which is called WaterGAP that computes current and future water availability worldwide. It partitions the tested global region in basins of various sizes which are forwarded to slaves for independent (fro other slaves) processing. Speed-up is limited by processing of the slave that takes the maximum of slaves' times. In order to deal with load imbalance, dynamic arrival of slaves has been adopted. The master assigns the tasks by size, from the largest task. Good allocation results in large basins being allocated to a process with many (powerful) processors, smaller basins to a process with fewer (weaker) processors. If a more powerful (in the aforementioned sense) slave arrives, the system can reassign a large basin. Furthermore, slave processes can dynamically split into either processes or threads for parallelization. The authors have concluded that MPI-2 provides needed support for these features apart from a scenario of sudden withdrawal of slaves in the context of proper finalization of an MPI application. No numerical results have been presented though.

MOST WIEDZY Downloaded from mostwiedzy.pl

In the case of OpenMP, implementations of master–slave and the producer–consumer pattern might share some elements. A buffer could be (but does not have to be) used for passing data between the master and slaves and is naturally used in producer–consumer implementations. In master–slave, the master would typically manage several data chunks ready to be distributed among slaves while in producer–consumer producer or producers will typically add one data chunk at a time to a buffer. Furthermore, in the producer–consumer pattern consumers do not return results to the producer(s). In the producer–consumer model we typically consider one or more producers and one or more consumers of data chunks. Data chunk production and consuming rates/speeds might differ, in which case a limited capacity buffer is used into which producer(s) inserts() data and consumer(s) fetches() data from for processing.

Book [1] contains three implementations of the master–slave paradigm in OpenMP. These include the designated-master, integrated-master and tasking, also considered in this work. Research presented in this paper extends directly those OpenMP implementations. Specifically, the paper extends the implementations with the dynamic-for version, as well as versions overlapping merging and data generation—tasking2 and dynamic-for2. Additionally, tests within this paper are run for a variety of thread affinity configurations, for various compute intensities as well as on four multi-core CPU models, of modern generations, including Kaby Lake, Coffee Lake, Broadwell and Skylake.

There have been several works focused on optimization of tasking in OpenMP that, as previously mentioned, can be used for implementation of master–slave. Specifically, in paper [23], authors proposed extensions of the tasking and related constructs with dependencies `produce` and `consume` which creates a multi-producer multi-consumer queue that is associated with a list item. Such a queue can be reused if it already exists. The life time of such a queue is linked to the life time of a parallel region that encompasses the construct. Such a construct can then be used for implementation of the master–slave model as well. In paper [24], the authors proposed an automatic correction algorithm meant for the OpenMP tasking model. It automatically generates correct task clauses and inserts appropriate task synchronization to maintain data dependence relationships. Authors of paper [25] show that when using OpenMP's tasks for stencil type of computations, when tasks are generated with `#pragma omp task` for a block of a 3D space, significant gains in performance are possible by adding block objects to locality queues from which a given thread executing a task dequeues blocks using an optimized policy.

3. Motivations, Application Model and Implementations

It should be emphasized that since the master–slave processing paradigm is widespread and at the same time multi-core CPUs are present in practically all desktops and workstations/cluster nodes thus it is important to investigate various implementations and determine preferred settings for such scenarios. At the same time, the processor families tested in this work are in fact representatives of the new generations CPUs in their respective CPU lines. The contribution of this work is experimental assessment of performance of proposed master–slave codes using OpenMP directives and library calls, compiled with gcc and `-fopenmp` flag for representative desktop and workstation systems with multicore CPUs listed in Table 1.

The model analyzed in this paper distinguishes the following conceptual steps, that are repeated:

1. Master generates a predefined number of data chunks from a data source if there is still data to be fetched from the data source.
2. Data chunks are distributed among slaves for parallel processing.
3. Results of individually processed data chunks are provided to the master for integration into a global result.

It should be noted that this model, assuming that the buffer size is smaller than the size of total input data, differs from a model in which all input data is generated at once by

MOST WIEDZY Downloaded from mostwiedzy.pl

the master. It might be especially well suited to processing, e.g., data from streams such as from the network, sensors or devices such as cameras, microphones etc.

3.1. Implementations of the Master–Slave Pattern with OpenMP

The OpenMP-based implementations of the analyzed master–slave model described in Section 3 and used for benchmarking are as follows:

1. designated-master (Figure 1)—direct implementation of master–slave in which a separate thread is performing the master's tasks of input data packet generation as well as data merging upon filling in the output buffer. The other launched threads perform slaves' tasks.
2. integrated-master (Figure 2)—modified implementation of the designated-master code. Master's tasks are moved to within a slave thread. Specifically, if a consumer thread has inserted the last result into the result buffer, it merges the results into a global shared result, clears its space and generates new data packets into the input buffer. If the buffer was large enough to contain all input data, such implementation would be similar to the all slave implementation shown in [19].
3. tasking (Figure 3)—code using the tasking construct. Within a region in which threads operate in parallel (created with `#pragma omp parallel`), one of the threads generates input data packets and launches tasks (in a loop) each of which is assigned processing of one data packet. These are assigned to the aforementioned threads. Upon completion of processing of all the assigned tasks, results are merged by the one designated thread, new input data is generated and the procedure is repeated.
4. tasking2—this version is an evolution of tasking. It potentially allows overlapping of generation of new data into the buffer and merging of latest results into the final result by the thread that launched computational tasks in version tasking. The only difference compared to the tasking version is that data generation is executed using `#pragma omp task`.
5. dynamic-for (Figure 4)—this version is similar to the tasking one with the exception that instead of tasks, in each iteration of the loop a function processing a given input data packet is launched. Parallelization of the for loop is performed with `#pragma omp for` with a dynamic chunk 1 size scheduling clause. Upon completion, output is merged, new input data is generated and the procedure is repeated.
6. dynamic-for2 (Figure 5)—this version is an evolution of dynamic-for. It allows overlapping of generation of new data into the buffer and merging of latest results into the final result through assignment of both operations to threads with various ids (such as 0 and 4 in the listing). It should be noted that ids of these threads can be controlled in order to make sure that these are threads running on different physical cores as was the case for the two systems tested in the following experiments.

For test purposes, all implementations used the buffer of 512 elements which is a multiple of the numbers of logical processors.

MOST WIEDZY Downloaded from mostwiedzy.pl

```
1    (...)
2    omp_init_lock(&inputlock);
3    omp_init_lock(&outputlock);
4    // firstly generate BUFFERSIZE data chunks of input data
5    lastgeneratedcount=generate_new_input(input);
6    init_final_output(&finaloutput);
7  #pragma omp parallel private(threadnumber,i) shared(work,input,output,finaloutput
       ,currentinputindex,currentoutputcount,lastgeneratedcount) num_threads(
       threadnum)
8    {
9      threadnumber=omp_get_thread_num();
10     if (threadnumber==0) { // master
11       long processedcount=0; // should finally reach CHUNKCOUNT
12       long merged=0;
13       while (processedcount<CHUNKCOUNT) {
14   omp_set_lock(&outputlock);
15   if (currentoutputcount==lastgeneratedcount) {
16             // process all available results
17     for(i=0;i<lastgeneratedcount;i++)
18       merge(&finaloutput,&(output[i]));
19     processedcount+=lastgeneratedcount;
20     currentoutputcount=0; merged=1;
21   }
22   omp_unset_lock(&outputlock);
23   omp_set_lock(&inputlock);
24   if ((currentinputindex>=lastgeneratedcount) && (merged)) { // generate new input
         data if the last results have been merged
25     if (processedcount<CHUNKCOUNT) {
26       lastgeneratedcount=generate_new_input(input);
27       currentinputindex=0; merged=0;
28     }
29   }
30   omp_unset_lock(&inputlock);
31       }
32 #pragma omp atomic write
33       work=0; // make slaves finish
34     } else { // slave
35       int processdata;
36       t_output result;
37       long myinputindex;
38       do {
39   processdata=0;
40   omp_set_lock(&inputlock);
41   myinputindex=currentinputindex;
42   if (currentinputindex<lastgeneratedcount) {
43     currentinputindex++; processdata=1;
44   }
45   omp_unset_lock(&inputlock);
46   if (processdata) {
47     // now process the input data chunk
48     result=process(&(input[myinputindex]));
49     // store the result in the output buffer
50     omp_set_lock(&outputlock);
51     if (currentoutputcount<BUFFERSIZE) {
52       output[currentoutputcount]=result;
53       currentoutputcount++;
54     }
55     omp_unset_lock(&outputlock);
56   }
57 #pragma omp atomic read
58   processdata=work;
59       } while (processdata);
60     }
61   }
62   print_final_output(&finaloutput);
63   (...)
```

Figure 1. Designated-master implementation.

MOST WIEDZY Downloaded from mostwiedzy.pl

```
1    (...)
2    omp_init_lock(&inputoutputlock);
3    // firstly generate BUFFERSIZE data chunks of input data
4    lastgeneratedcount=generate_new_input(input);
5    init_final_output(&finaloutput);
6  #pragma omp parallel private(i) shared(input,output,finaloutput,currentinputindex
       ,currentoutputcount,lastgeneratedcount,processedcount) num_threads(threadnum)
7    {
8      // each thread acts as a slave
9      int processdata;
10     t_output result;
11     long myinputindex;
12     int finish;
13     do {
14       processdata=0;
15       finish=0;
16       omp_set_lock(&inputoutputlock);
17       if (processedcount<CHUNKCOUNT) {
18         myinputindex=currentinputindex;
19         if (currentinputindex<lastgeneratedcount) {
20        currentinputindex++;
21    processdata=1;
22         }
23       } else finish=1;
24       omp_unset_lock(&inputoutputlock);
25
26       if (processdata) {
27  // now process the input data chunk
28  result=process(&(input[myinputindex]));
29  // store the result in the output buffer
30  omp_set_lock(&inputoutputlock);
31  if (currentoutputcount<BUFFERSIZE) {
32    output[currentoutputcount]=result;
33    currentoutputcount++;
34  }
35  if (currentoutputcount==lastgeneratedcount) { // process all available results
36    for(i=0;i<lastgeneratedcount;i++)
37      merge(&finaloutput,&(output[i]));
38    processedcount+=lastgeneratedcount;
39    currentoutputcount=0;
40
41    if (processedcount<CHUNKCOUNT) {
42      lastgeneratedcount=generate_new_input(input);
43      currentinputindex=0;
44    }
45  }
46  omp_unset_lock(&inputoutputlock);
47       }
48     } while (!finish);
49   }
50   print_final_output(&finaloutput);
51   (...)
```

Figure 2. Integrated-master implementation.

```
1    (...)
2    init_final_output(&finaloutput);
3  #pragma omp parallel private(i,myinputindex) shared(input,output,finaloutput,
       lastgeneratedcount) num_threads(threadnum)
4    {
5  #pragma omp single
6      {
7        long processedcount=0;
8        do {
9   lastgeneratedcount=generate_new_input(input);
10  // now create tasks that will deal with data packets
11  for(myinputindex=0;myinputindex<lastgeneratedcount;myinputindex++)
12    {
13 #pragma omp task firstprivate(myinputindex) shared(input,output)
14      {
15        // now each task is processed independently and can store its result into
              an appropriate buffer
16        output[myinputindex]=process(&(input[myinputindex]));
17      }
18    }
19  // wait for tasks
20 #pragma omp taskwait
21  // now merge results
22  for(i=0;i<lastgeneratedcount;i++)
23    merge(&finaloutput,&(output[i]));
24  processedcount+=lastgeneratedcount;
25        } while (processedcount<CHUNKCOUNT);
26      }
27   }
28    print_final_output(&finaloutput);
29    (...)
```

Figure 3. Tasking implementation.

```
1    (...)
2    init_final_output(&finaloutput);
3    int work=1;
4  #pragma omp parallel private(i,myinputindex) shared(work,input,output,finaloutput
       ,lastgeneratedcount) num_threads(threadnum)
5    {
6      long processedcount=0;
7      int processdata=1;
8      do {
9  #pragma omp master
10       {
11  lastgeneratedcount=generate_new_input(input);
12  processedcount+=lastgeneratedcount;
13  // master checks if there is more data to process
14  if (processedcount>=CHUNKCOUNT) {
15    processdata=0;
16 #pragma omp atomic write
17    work=0; // make slaves finish
18  }
19       }
20 #pragma omp barrier
21       // now slaves can read the termination flag
22 #pragma omp atomic read
23  processdata=work;
24 #pragma omp for schedule(dynamic,1)
25       // now create tasks that will deal with data packets
26       for(myinputindex=0;myinputindex<lastgeneratedcount;myinputindex++)
27  {
28    output[myinputindex]=process(&(input[myinputindex]));
29  }
30 #pragma omp master
31         {
32    // now merge results
33    for(i=0;i<lastgeneratedcount;i++)
34    merge(&finaloutput,&(output[i]));
35         }
36     } while (processdata);
37   }
38   print_final_output(&finaloutput);
39   (...)
```

Figure 4. Dynamic-for implementation.

MOST WIEDZY Downloaded from mostwiedzy.pl

```
1    (...)
2    init_final_output(&finaloutput);
3    int work=1;
4  #pragma omp parallel private(i,myinputindex) shared(work,input,output,finaloutput
       ,lastgeneratedcount) num_threads(threadnum)
5    {
6      long processedcount=0;
7      int processdata=1;
8      int mythreadid=omp_get_thread_num();
9      long lastgeneratedcounttemp=0;
10     if (mythreadid==0)
11       {
12   lastgeneratedcount=generate_new_input(input);
13   processedcount+=lastgeneratedcount;
14   // master checks if there is more data to process
15   if (processedcount>=CHUNKCOUNT) {
16     processdata=0;
17 #pragma omp atomic write
18     work=0; // make slaves finish
19   }
20
21       }
22 #pragma omp barrier
23     do {
24       lastgeneratedcounttemp=lastgeneratedcount;
25 #pragma omp barrier
26       // now slaves can read the termination flag
27 #pragma omp atomic read
28       processdata=work;
29 #pragma omp for schedule(dynamic,1)
30       // now create tasks that will deal with data packets
31       for(myinputindex=0;myinputindex<lastgeneratedcount;myinputindex++)
32       {
33   output[myinputindex]=process(&(input[myinputindex]));
34       }
35       if (mythreadid==0) {
36   if (processdata) { // generate new data only if this is not the last iteration
37     lastgeneratedcount=generate_new_input(input);
38     processedcount+=lastgeneratedcount;
39     // master checks if there is more data to process
40     if (processedcount>=CHUNKCOUNT) {
41 #pragma omp atomic write
42       work=0; // make slaves finish
43     }
44   }
45       }
46       else if (mythreadid==4) {
47   // now merge results
48   for(i=0;i<lastgeneratedcounttemp;i++)
49   merge(&finaloutput,&(output[i]));
50       }
51     } while (processdata);
52   }
53    print_final_output(&finaloutput);
54    (...)
```

Figure 5. Dynamic-for2 implementation.

4. Experiments

4.1. Parametrized Irregular Testbed Applications

The following two applications are irregular in nature which results in various execution times per data chunk and subsequently exploits the dynamic load balancing capabilities of the tested master–slave implementations.

4.1.1. Parallel Adaptive Quadrature Numerical Integration

The first, compute-intensive, application, is numerical integration of any given function. For benchmarking, integration of $f(x) = x \cdot sin^2(x^2)$ was run over the [0, 100] range. The range was partitioned into 100,000 subranges which were regarded as data chunks in the processing scheme. Each subrange was then integrated (by a slave) by using the following adaptive quadrature [26] and recursive technique for a given range [a, b] being considered:

1. if the area of triangle $(a, f(a)), (b, f(b)), (\frac{a+b}{2}, f(\frac{a+b}{2}))$ is smaller than 10^{-k}/partitioning coefficient (k=18) then the sum of areas of two trapezoids

MOST WIEDZY Downloaded from mostwiedzy.pl

$(a,0),(\frac{a+b}{2},0),(\frac{a+b}{2},f(\frac{a+b}{2})),(a,f(a))$ and $(\frac{a+b}{2},0),(b,0),(b,f(b)),(\frac{a+b}{2},$ $f(\frac{a+b}{2}))$ is returned as a result,

2. otherwise, recursive partitioning into two subranges $(a,\frac{a+b}{2})$ and $(\frac{a+b}{2},b)$ is performed and the aforementioned procedure is repeated for each of these until the condition is met.

This way increasing the partitioning coefficient increases accuracy of computations and consequently increases the compute to synchronization ratio. Furthermore, this application does not require large size memory and is not memory bound.

4.1.2. Parallel Image Recognition

In contrast to the previous application, parallel image recognition was used as a benchmark that requires much memory and frequent memory reads. Specifically, the goal of the application is to search for at least one occurrence of a template (sized TEMPLATEXSIZExTEMPLATEYSIZE in pixels) within an image (sized IMAGEXSIZExIMAGEYSIZE).

In this case, the initial image is partitioned and within each chunk, a part of the initial image of size (TEMPLATEXSIZE + BLOCKXSIZE)x
(TEMPLATEYSIZE + BLOCKYSIZE) is searched for occurrence of the template. In the actual implementation values of IMAGEXSIZE = IMAGEYSIZE = 20,000, BLOCKXSIZE = BLOCKYSIZE = 20, TEMPLATEXSIZE =TEMPLATEYSIZE = 500 in pixels were used.

The image was initialized with every third row and every third column having pixels not matching the template. This results in earlier termination of search for template, also depending on the starting search location in the initial image which results in various search times per chunk.

In the case of this application a compute coefficient reflects how many passes over the initial image are performed. In actual use cases it might correspond to scanning slightly updated images in a series (e.g., satellite images or images of location taken with a drone) for objects. On the other hand, it allows to simulate scenarios of various relative compute to memory access and synchronization overheads for various systems.

4.2. Testbed Environment and Methodology of Tests

Experiments were performed on two systems typical of a modern desktop and workstation systems with specifications outlined in Table 1.

Table 1. Testbed configurations.

Testbed	**1**	**2**
CPUs	Intel(R) Core(TM) i7-7700 CPU 3.60 GHz Kaby Lake, 8 MB cache	2 × Intel(R) Xeon(R) CPU E5-2620 v4 2.10 GHz Broadwell, 20 MB cache per CPU
CPUs— total number of physical/logical processors	4/8	16/32
System memory size (RAM) [GB]	16 GB	128 GB
Operating system	Ubuntu 18.04.1 LTS	Ubuntu 20.04.1 LTS
Compiler/version	gcc version 9.3.0 (Ubuntu 9.3.0-11ubuntu0 18.04.1),	gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1 20.04),

The following combinations of tests were performed: {code implementation} × {range of thread counts} × {affinity setting} × {partitioning coefficients: 1, 8, 32}. The range of thread counts tested depends on the implementation and varied as follows, based on preliminary tests that identified the most interesting values based on most promising execution times, where npl means the number of logical processors: for designated-master these were $npl/2$, $1+npl/2$, npl and $1+npl$, for all other versions the following were tested: $npl/4$, $npl/2$, npl and $2\cdot npl$. Thread affinity settings were imposed with environment variables `OMP_PLACES` and `OMP_PROC_BIND` [27,28]. Specifically, the following combinations

MOST WIEDZY Downloaded from mostwiedzy.pl

were tested independently: default (no additional affinity settings) marked with default, `OMP_PROC_BIND=false` which turns off thread affinity (marked in results as noprocbind), `OMP_PLACES=cores` and `OMP_PROC_BIND=close` marked with corclose, `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` marked with corspread, `OMP_PLACES=threads` and `OMP_PROC_BIND=close` marked with thrclose, `OMP_ PLACES=sockets` without setting `OMP_PROC_BIND` marked with sockets which defaults to true if `OMP_PLACES` is set for gcc (https://gcc.gnu.org/onlinedocs/gcc-9.3.0/libgomp/OMP_005fPROC_005fBIND.html (accessed on 16 April 2021)). If `OMP_PROC_BIND` equals `true` then behavior is implementation defined and thus the above concrete settings were tested.In the experiments the code was tested with compilation flags `-O3` and also `-O3 -march=native`. Best values are reported for each configuration, an average value out of 20 runs is presented along with corresponding standard deviation.

4.3. Results

Since all combinations of tested configurations resulted in a very large number of execution times, we present best results as follows. For each partitioning coefficient separately for numerical integration and compute coefficient for image recognition and for each code implementation 3 best results with a configuration description are presented in Tables 2 and 3 for numerical integration as well as in Tables 4 and 5 for image recognition, along with the standard deviation computed from the results. Consequently, it is possible to identify how code versions compare to each other and how configurations affect execution times.

Additionally, for the coefficients, execution times and corresponding standard deviation values are shown for various numbers of threads. These are presented in Figures 6 and 7 for numerical integration as well as in Figures 8 and 9 for image recognition.

Table 2. Numerical integration—system 1 results.

Part. Coeff.	Version	Time 1/std dev/Affinity/Number of Threads	Time 2/std dev/Affinity/Number of Threads	Time 3/std dev/Affinity/Number of Threads
1	integrated-master	18.555/0.062/thrclose/8	18.610/0.068/corspread/8	18.641/0.086/corclose/8
	designated-master	21.088/0.078/corspread/8	21.098/0.090/default/8	21.106/0.096/noprocbind/8
	tasking	18.363/0.047/noprocbind/16	18.416/0.094/default/16	18.595/0.071/thrclose/8
	tasking2	18.394/0.088/noprocbind/16	18.411/0.093/default/16	18.654/0.092/thrclose/8
	dynamic-for	18.389/0.079/default/16	18.428/0.105/noprocbind/16	18.554/0.073/corclose/16
	dynamic-for2	18.399/0.093/default/16	18.416/0.101/noprocbind/16	18.572/0.073/corclose/16
8	integrated-master	27.333/0.105/thrclose/8	27.341/0.106/default/8	27.373/0.084/noprocbind/8
	designated-master	30.885/0.102/corspread/8	30.898/0.111/thrclose/8	30.956/0.130/noprocbind/8
	tasking	26.844/0.081/default/16	26.898/0.146/noprocbind/16	27.325/0.116/default/8
	tasking2	26.865/0.131/default/16	26.901/0.161/noprocbind/16	27.299/0.134/thrclose/8
	dynamic-for	26.865/0.105/default/16	26.899/0.155/noprocbind/16	27.217/0.121/corspread/16
	dynamic-for2	26.830/0.073/noprocbind/16	26.930/0.158/default/16	27.204/0.115/corclose/16
32	integrated-master	34.492/0.157/thrclose/8	34.526/0.137/corspread/8	34.555/0.202/corclose/8
	designated-master	39.005/0.149/thrclose/8	39.015/0.199/corclose/8	39.039/0.174/default/8
	tasking	33.816/0.151/noprocbind/16	33.889/0.333/default/16	34.356/0.149/noprocbind/8
	tasking2	33.828/0.174/noprocbind/16	33.838/0.152/default/16	34.340/0.148/thrclose/8
	dynamic-for	33.781/0.165/noprocbind/16	33.808/0.148/default/16	34.354/0.165/thrclose/16
	dynamic-for2	33.826/0.180/noprocbind/16	33.860/0.155/default/16	34.260/0.127/corclose/16

MOST WIEDZY Downloaded from mostwiedzy.pl

Table 3. Numerical integration—system 2 results.

Part. Coeff.	Version	Time 1/std dev/Affinity/Number of Threads	Time 2/std dev/Affinity/Number of Threads	Time 3/std dev/Affinity/Number of Threads
1	integrated-master	9.158/0.117/corspread/32	9.201/0.145/thrclose/32	9.214/0.217/sockets/32
	designated-master	9.585/0.149/corclose/33	9.601/0.197/thrclose/33	9.638/0.122/default/33
	tasking	8.567/0.017/default/64	8.585/0.027/noprocbind/64	8.664/0.025/sockets/64
	tasking2	8.599/0.033/default/64	8.602/0.025/noprocbind/64	8.677/0.026/sockets/64
	dynamic-for	8.584/0.025/noprocbind/64	8.584/0.032/default/64	8.649/0.024/sockets/64
	dynamic-for2	8.570/0.024/default/64	8.573/0.021/noprocbind/64	8.636/0.024/sockets/64
8	integrated-master	13.718/0.127/corclose/32	13.748/0.182/corspread/32	13.770/0.111/default/32
	designated-master	14.402/0.105/corclose/33	14.447/0.529/thrclose/32	14.481/0.677/sockets/32
	tasking	12.724/0.034/default/64	12.727/0.040/noprocbind/64	12.776/0.038/sockets/64
	tasking2	12.749/0.044/default/64	12.771/0.035/noprocbind/64	12.796/0.044/sockets/64
	dynamic-for	12.792/0.041/default/64	12.796/0.033/noprocbind/64	12.845/0.031/sockets/64
	dynamic-for2	12.731/0.031/default/64	12.753/0.040/noprocbind/64	12.811/0.047/sockets/64
32	integrated-master	17.471/0.080/corspread/32	17.486/0.105/corclose/32	17.551/0.152/thrclose/32
	designated-master	18.359/0.839/corspread/32	18.423/0.447/default/32	18.431/0.205/corclose/33
	tasking	16.116/0.051/noprocbind/64	16.120/0.055/sockets/64	16.175/0.420/default/64
	tasking2	16.119/0.039/default/64	16.142/0.062/noprocbind/64	16.157/0.042/sockets/64
	dynamic-for	16.181/0.049/default/64	16.210/0.043/noprocbind/64	16.228/0.046/sockets/64
	dynamic-for2	16.116/0.025/default/64	16.119/0.043/noprocbind/64	16.152/0.038/sockets/64

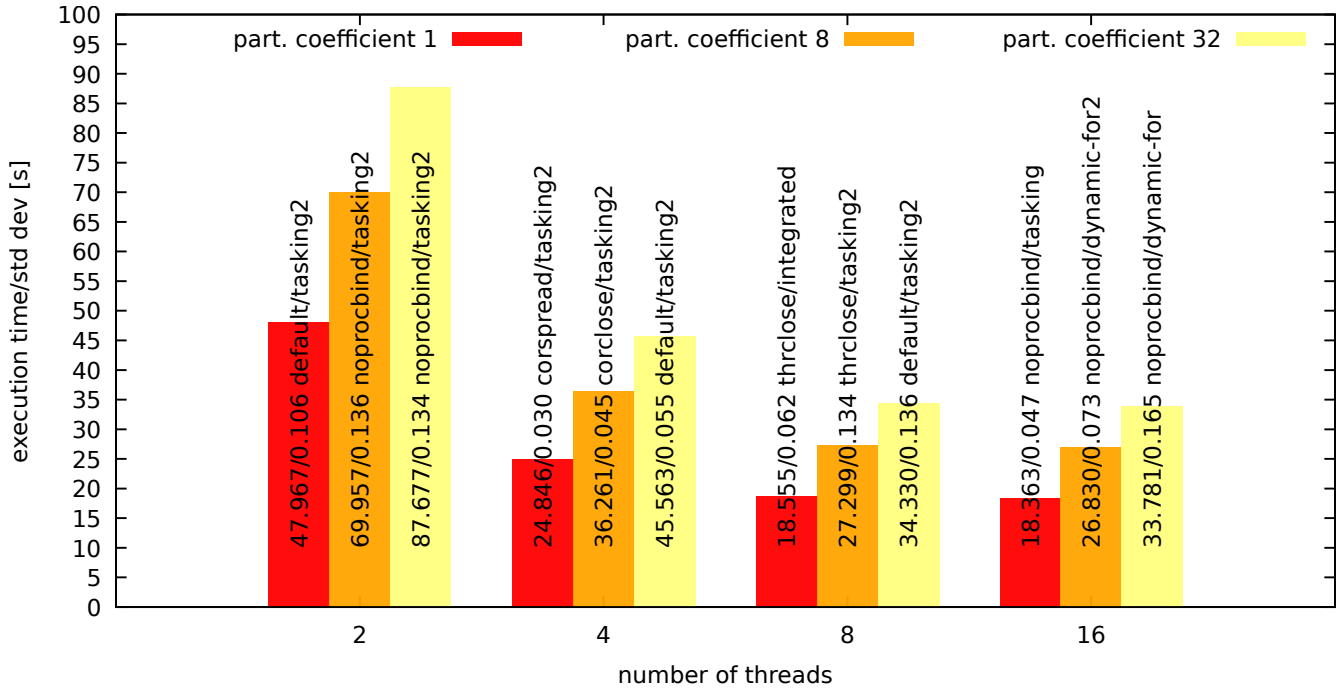

Figure 6. Numerical integration—system 1 results for various numbers of threads.

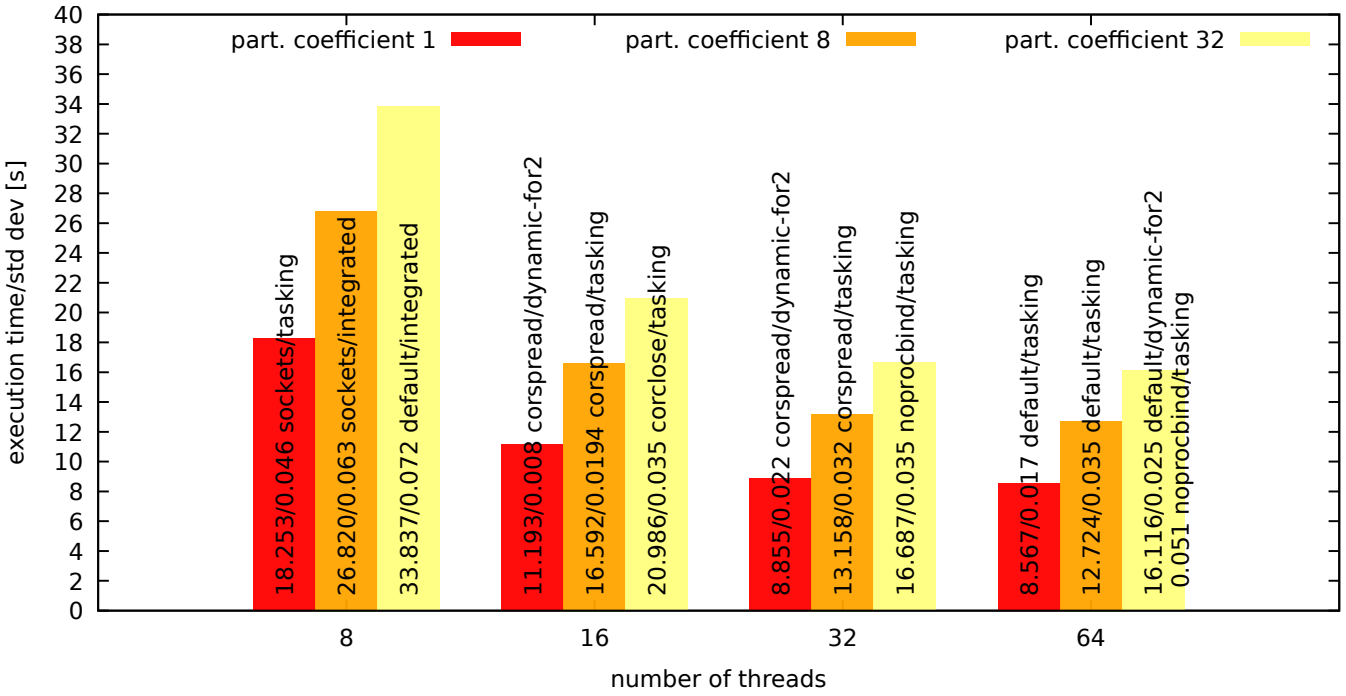

Figure 7. Numerical integration—system 2 results for various numbers of threads.

Table 4. Image recognition—system 1 results.

Comp. Coeff.	Version	Time 1/std dev/Affinity/Number of Threads	Time 2/std dev/Affinity/Number of Threads	Time 3/std dev/Affinity/Number of Threads
2	integrated-master	9.530/0.2104/noprocbind/8	9.561/0.173/default/8	9.578/0.183/thrclose/8
	designated-master	10.388/0.125/thrclose/8	10.434/0.179/noprocbind/8	10.450/0.222/default/8
	tasking	9.576/0.175/default/8	9.622/0.166/noprocbind/8	9.697/0.188/corclose/8
	tasking2	12.762/0.059/noprocbind/8	12.777/0.093/thrclose/8	12.782/0.081/default/8
	dynamic-for	9.389/0.131/thrclose/8	9.392/0.156/noprocbind/8	9.403/0.151/default/8
	dynamic-for2	9.378/0.135/thrclose/8	9.395/0.165/default/8	9.446/0.176/default/16
4	integrated-master	18.406/0.297/noprocbind/8	18.428/0.329/corclose/8	18.492/0.352/default/8
	designated-master	20.175/0.196/corspread/8	20.219/0.305/default/8	20.404/0.367/noprocbind/8
	tasking	18.505/0.428/noprocbind/8	18.514/0.308/thrclose/8	18.540/0.353/default/8
	tasking2	24.935/0.154/noprocbind/8	24.940/0.150/corspread/8	24.967/0.244/thrclose/8
	dynamic-for	18.332/0.264/noprocbind/8	18.332/0.475/default/8	18.405/0.442/corspread/8
	dynamic-for2	18.282/0.229/corspread/8	18.318/0.407/thrclose/8	18.367/0.408/default/8
8	integrated-master	35.995/0.678/noprocbind/8	36.096/0.726/default/8	36.282/0.612/thrclose/8
	designated-master	39.969/0.526/default/8	40.120/0.595/corclose/8	40.163/0.623/thrclose/8
	tasking	36.223/0.718/noprocbind/8	36.307/0.691/corspread/8	36.372/0.664/thrclose/8
	tasking2	49.418/0.225/default/8	49.438/0.411/noprocbind/8	49.444/0.326/corspread/8
	dynamic-for	35.852/0.503/default/8	36.018/0.596/corspread/16	36.129/0.597/noprocbind/16
	dynamic-for2	35.969/0.462/thrclose/8	36.099/0.669/default/8	36.190/0.675/noprocbind/8

Table 5. Image recognition—system 2 results.

Comp. Coeff.	Version	Time 1/std dev/Affinity/Number of Threads	Time 2/std dev/Affinity/Number of Threads	Time 3/std dev/Affinity/Number of Threads
2	integrated-master	6.406/0.321/thrclose/32	6.880/0.918/default/32	7.002/0.738/corclose/32
	designated-master	6.283/0.311/sockets/33	6.644/0.364/noprocbind/33	6.697/0.463/default/33
	tasking	6.164/0.145/corclose/64	6.223/0.181/corspread/64	6.249/0.117/sockets/64
	tasking2	5.981/0.208/corclose/64	5.995/0.165/corspread/64	5.997/0.067/sockets/64
	dynamic-for	5.705/0.208/default/32	5.722/0.105/sockets/64	5.739/0.088/corclose/32
	dynamic-for2	5.682/0.072/sockets/32	5.697/0.055/noprocbind/32	5.709/0.099/default/32
4	integrated-master	11.583/0.564/noprocbind/32	11.661/0.218/sockets/32	11.716/0.420/corclose/32
	designated-master	11.808/1.572/corclose/32	11.857/1.097/sockets/33	11.878/0.803/noprocbind/33
	tasking	10.848/0.085/default/32	10.889/0.128/corclose/32	10.903/0.142/sockets/32
	tasking2	10.460/0.141/sockets/32	10.472/0.145/corspread/32	10.485/0.170/default/32
	dynamic-for	10.625/0.140/default/32	10.629/0.133/corclose/32	10.635/0.161/noprocbind/32
	dynamic-for2	10.585/0.150/sockets/32	10.598/0.100/noprocbind/32	10.610/0.140/default/32
8	integrated-master	20.556/0.620/noprocbind/32	20.595/0.708/corclose/32	20.738/0.861/corspread/32
	designated-master	20.705/0.836/default/33	20.924/4.271/sockets/32	21.224/0.987/noprocbind/33
	tasking	20.014/0.197/sockets/32	20.054/0.201/corclose/32	20.076/0.235/corspread/32
	tasking2	19.120/0.305/noprocbind/32	19.152/0.187/sockets/32	19.240/0.292/corspread/32
	dynamic-for	19.758/0.193/default/32	19.825/0.210/thrclose/32	19.828/0.219/corspread/32
	dynamic-for2	19.816/0.229/noprocbind/32	19.828/0.249/default/32	19.863/0.256/thrclose/32

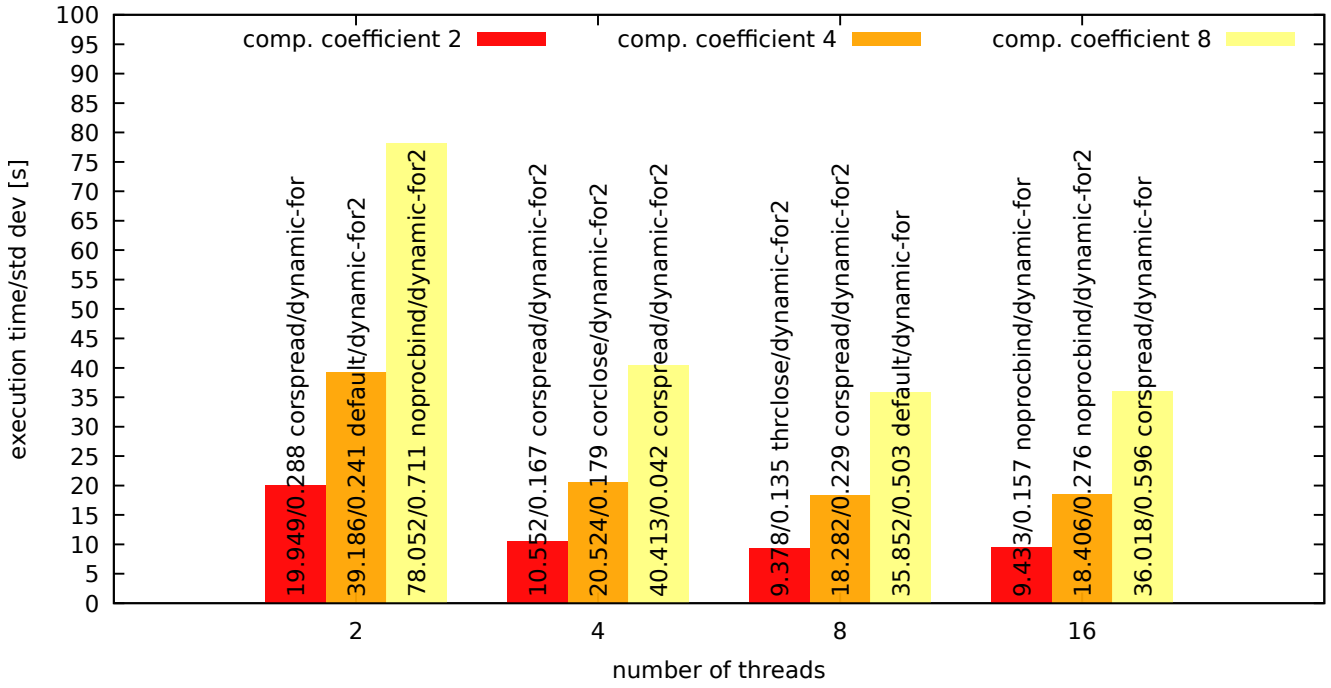

Figure 8. Image recognition—system 1 results for various numbers of threads.

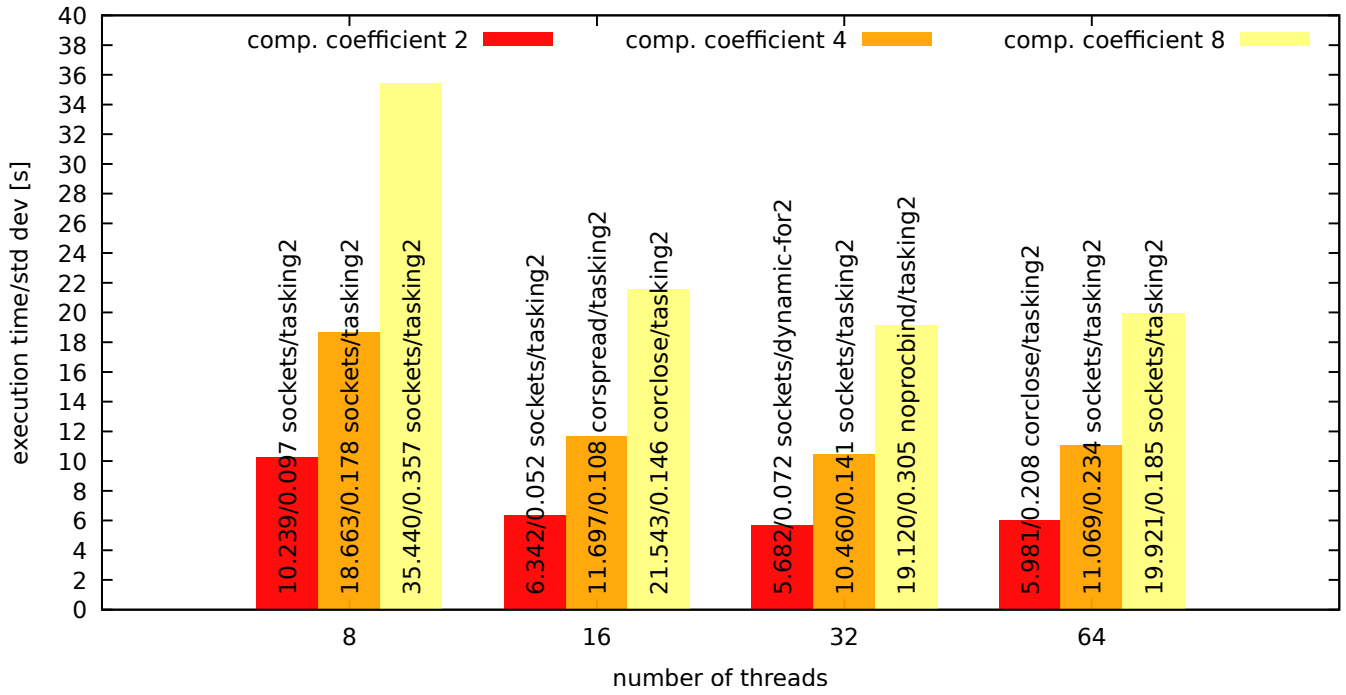

Figure 9. Image recognition—system 2 results for various numbers of threads.

4.4. Observations and Discussion

4.4.1. Performance

From the performance point of view, based on the results the following observations can be drawn and subsequently be generalized:

1. For numerical integration, best implementations are tasking and dynamic-for2 (or dynamic-for for system 1) with practically very similar results. These are very closely followed by tasking2 and dynamic-for and then by visibly slower integrated-master and designated-master.
2. For image recognition best implementations for system 1 are dynamic-for2/dynamic-for and integrated-master with very similar results, followed by tasking, designated-master and tasking2. For system 2, best results are shown by dynamic-for2/dynamic-for and tasking2, followed by tasking and then by visibly slower integrated-master and designated-master.
3. For system 2, we can see benefits from overlapping for dynamic-for2 over dynamic-for for numerical integration and for both tasking2 over tasking, as well as dynamic-for2 over dynamic-for for image recognition. The latter is expected as those configurations operate on considerably larger data and memory access times constitute a larger part of the total execution time, compared to integration.
4. For the compute intensive numerical integration example we see that best results were generally obtained for oversubscription, i.e., for tasking* and dynamic-for* best numbers of threads were 64 rather than generally 32 for system 2 and 16 rather than 8 for system 1. The former configurations apparently allow to mitigate idle time without the accumulated cost of memory access in the case of oversubscription.
5. In terms of thread affinity, for the two applications best configurations were measured for default/noprocbind for numerical integration for both systems and for thrclose/-corspread for system 1 and sockets for system 2 for smaller compute coefficients and default for system 1 and noprocbind for system 2 for compute coefficient 8.
6. For image recognition, configurations generally show visibly larger standard deviation than for numerical integration, apparently due to memory access impact.

7. We can notice that relative performance of the two systems is slightly different for the two applications. Taking into account best configurations, for numerical integration system 2's times are approx. 46–48% of system 1's times while for image recognition system 2's times are approx. 53–61% of system 1's times, depending on partitioning and compute coefficients.
8. We can assess gain from HyperThreading for the two applications and the two systems (between 4 and 8 threads for system 1 and between 16 and 32 threads for system 2) as follows: for numerical integration and system 1 it is between 24.6% and 25.3% for the coefficients tested, for system 2 it is between 20.4% and 20.9%; for image recognition and system 1, it is between 10.9% and 11.3% and similarly for system 2 between 10.4% and 11.3%.
9. We can see that ratios of best system 2 to system 1 times for image recognition are approx. 0.61 for coefficient 2, 0.57 for coefficient 4 and 0.53 for coefficient 8 which means that results for system 2 for this application get relatively better compared to system 1's. As outlined in Table 1, system 2 has larger cache and for subsequent passes more data can reside in the cache. This behavior can also be seen when results for 8 threads are compared—for coefficients 2 and 4 system 1 gives shorter times but for coefficient 8 system 2 is faster.
10. integrated-master is relatively better compared to the best configuration for system 1 as opposed to system 2—in this case, the master's role can be taken by any thread, running on one of the 2 CPUs.

The bottom line, taking into consideration the results, is that preferred configurations are tasking and dynamic-for based ones, with preferring thread oversubscription (2 threads per logical processor) for the compute intensive numerical integration and 1 thread per logical processor for memory requiring image recognition. In terms of affinity, default/noprocbind are to be preferred for numerical integration for both systems and thrclose/corspread for system 1 and sockets for system 2 for smaller compute coefficients and default for system 1 and noprocbind for system 2 for compute coefficient 8.

4.4.2. Ease of Programming

Apart from the performance of the proposed implementations, ease of programming can be assessed in terms of the following aspects:

1. code length—the order from the shortest to the longest version of the code is as follows: tasking, dynamic-for, tasking2, integrated-master, dynamic-for2 and designated-master,
2. the numbers of OpenMP directives and functions. In this case the versions can be characterized as follows:
    - designated-master—3 directives and 13 function calls;
    - integrated-master—1 directive and 6 function calls;
    - tasking—4 directives and 0 function calls;
    - tasking2—6 directives and 0 function calls;
    - dynamic-for—7 directives and 0 function calls;
    - dynamic-for2—7 directives and 1 function call,

    which makes tasking the most elegant and compact solution.
3. controlling synchronization—from the programmer's point of view this seems more problematic than the code length, specifically how many distinct thread codes' points need to synchronize explicitly in the code. In this case, the easiest code to manage is tasking/tasking2 as synchronization of independently executed tasks is performed in a single thread. It is followed by integrated-master which synchronizes with a lock in two places and dynamic-for/dynamic-for2 which require thread synchronization within `#pragma omp parallel`, specifically using atomics and designated-master which uses two locks, each in two places. This aspect potentially indicates how prone to errors each of these implementations can be for a programmer.

MOST WIEDZY Downloaded from mostwiedzy.pl

5. Conclusions and Future Work

Within the paper, we compared six different implementations of the master–slave paradigm in OpenMP and tested relative performances of these solutions using a typical desktop system with 1 multi-core CPU—Intel i7 Kaby Lake and a workstation system with 2 multi-core CPUs—Intel Xeon E5 v4 Broadwell CPUs.

Tests were performed for irregular numerical integration and irregular image recognition with three various compute intensities and for various thread affinities, compiled with the popular `gcc` compiler. Best results were generally obtained for OpenMP task and dynamic for based construct implementations, either with thread oversubscription (numerical integration) or without oversubscription (image recognition) for the aforementioned applications.

Future work includes investigation of aspects such as the impact of buffer length and false sharing on the overall performance of the model, as well as performing tests using other compilers and libraries. Furthermore, tests with a different compiler and OpenMP library such as using, e.g., `icc -openmp` would be practical and interesting for their users. Another research direction relates to consideration of potential performance-energy aspects of implementations in the context of CPUs used and configurations, also when using power capping as an extension of previous works in this field [29–31]. Finally, investigation of performance of basic OpenMP constructs for modern multi-core systems and compilers is of interest, as an extension of previous works such as [32,33].

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Czarnul, P. *Parallel Programming for Modern High Performance Computing Systems*; Chapman and Hall/CRC Press/Taylor & Francis: Boca Raton, FL, USA 2018.
2. Klemm, M.; Supinski, B.R. (Eds.) *OpenMP Application Programming Interface Specification Version 5.0*; OpenMP Architecture Review Board: Chicago, IL, USA, 2019; ISBN 978-1795759885.
3. Ohberg, T.; Ernstsson, A.; Kessler, C. Hybrid CPU–GPU execution support in the skeleton programming framework SkePU. *J. Supercomput.* **2019**, *76*, 5038–5056. [CrossRef]
4. Ernstsson, A.; Li, L.; Kessler, C. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *Int. J. Parallel Program.* **2018**, *46*, 62–80. [CrossRef]
5. Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P.A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. Pract. Exp.* **2011**, *23*, 187–198. [CrossRef]
6. Pop, A.; Cohen, A. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.* **2013**, *9*. [CrossRef]
7. Pop, A.; Cohen, A. A Stream-Computing Extension to OpenMP. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, Crete, Greece, 24–26 January 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 5–14. [CrossRef]
8. Seo, S.; Amer, A.; Balaji, P.; Bordage, C.; Bosilca, G.; Brooks, A.; Carns, P.; Castelló, A.; Genet, D.; Herault, T.; et al. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 512–526. [CrossRef]
9. Pereira, A.D.; Ramos, L.; Góes, L.F.W. PSkel: A stencil programming framework for CPU-GPU systems. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 4938–4953. [CrossRef]
10. Balart, J.; Duran, A.; Gonzalez, M.; Martorell, X.; Ayguade, E.; Labarta, J. Skeleton driven transformations for an OpenMP compiler. In Proceedings of the 11th Workshop on Compilers for Parallel Computers (CPC 04), Chiemsee, Germany, 7–9 July 2004; pp. 123–134.
11. Bondhugula, U.; Baskaran, M.; Krishnamoorthy, S.; Ramanujam, J.; Rountev, A.; Sadayappan, P. *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*; Compiler Construction; Hendren, L., Ed.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 132–146.
12. Czarnul, P. Parallelization of Divide-and-Conquer Applications on Intel Xeon Phi with an OpenMP Based Framework. In *Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology—ISAT 2015—Part III*; Swiatek, J.; Borzemski, L.; Grzech, A.; Wilimowska, Z., Eds.; Springer: Cham, Switzerland, 2016; pp. 99–111.
13. Fernández, A.; Beltran, V.; Martorell, X.; Badia, R.M.; Ayguadé, E.; Labarta, J. Task-Based Programming with OmpSs and Its Application. In *Euro-Par 2014: Parallel Processing Workshops*; Lopes, L., Žilinskas, J., Costan, A., Cascella, R.G., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., et al., Eds.; Springer: Cham, Switzerland, 2014; pp. 601–612.

MOST WIEDZY Downloaded from mostwiedzy.pl

14. Vidal, R.; Casas, M.; Moretó, M.; Chasapis, D.; Ferrer, R.; Martorell, X.; Ayguadé, E.; Labarta, J.; Valero, M. Evaluating the Impact of OpenMP 4.0 Extensions on Relevant Parallel Workloads. In *OpenMP: Heterogenous Execution and Data Movements*; Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S., Eds.; Springer: Cham, Switzerland, 2015; pp. 60–72.
15. Ciesko, J.; Mateo, S.; Teruel, X.; Beltran, V.; Martorell, X.; Badia, R.M.; Ayguadé, E.; Labarta, J. Task-Parallel Reductions in OpenMP and OmpSs. In *Using and Improving OpenMP for Devices, Tasks, and More*; DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S., Eds.; Springer: Cham, Switzerland, 2014; pp. 1–15.
16. Bosch, J.; Vidal, M.; Filgueras, A.; Álvarez, C.; Jiménez-González, D.; Martorell, X.; Ayguadé, E. Breaking master–slave Model between Host and FPGAs. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20, San Diego, CA, USA, 22–26 February 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 419–420. [CrossRef]
17. Dongarra, J.; Gates, M.; Haidar, A.; Kurzak, J.; Luszczek, P.; Wu, P.; Yamazaki, I.; Yarkhan, A.; Abalenkovs, M.; Bagherpour, N.; et al. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Softw.* **2019**, *45*. [CrossRef]
18. Valero-Lara, P.; Catalán, S.; Martorell, X.; Usui, T.; Labarta, J. sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs Library). *J. Parallel Distrib. Comput.* **2020**, *138*, 153–171. [CrossRef]
19. Schmider, H. *Shared-Memory Programming Programming with OpenMP*; Ontario HPC Summer School, Centre for Advance Computing, Queen's University: Kingston, ON, Canada 2018.
20. Hadjidoukas, P.E.; Polychronopoulos, E.D.; Papatheodorou, T.S. OpenMP for Adaptive master–slave Message Passing Applications. In *High Performance Computing*; Veidenbaum, A., Joe, K., Amano, H., Aiso, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 540–551.
21. Liu, G.; Schmider, H.; Edgecombe, K.E. A Hybrid Double-Layer master–slave Model For Multicore-Node Clusters. *J. Phys. Conf. Ser.* **2012**, *385*, 12011. [CrossRef]
22. Leopold, C.; Süß, M. Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*; Mohr, B., Träff, J.L., Worringen, J., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2006.
23. Scogland, T.; de Supinski, B. A Case for Extending Task Dependencies. In *OpenMP: Memory, Devices, and Tasks*; Maruyama, N., de Supinski, B.R., Wahib, M., Eds.; Springer: Cham, Switzerland, 2016; pp. 130–140.
24. Wang, C.K.; Chen, P.S. Automatic scoping of task clauses for the OpenMP tasking model. *J. Supercomput.* **2015**, *71*, 808–823. [CrossRef]
25. Wittmann, M.; Hager, G. A Proof of Concept for Optimizing Task Parallelism by Locality Queues. *arXiv* **2009**, arXiv:0902.1884.
26. Czarnul, P. Programming, Tuning and Automatic Parallelization of Irregular Divide-and-Conquer Applications in DAMPVM/-DAC. *Int. J. High Perform. Comput. Appl.* **2003**, *17*, 77–93. [CrossRef]
27. Eijkhout, V.; van de Geijn, R.; Chow E. Introduction to High Performance Scientific Computing; lulu.com. 2011. Available online: http://www.tacc.utexas.edu/$\sim$eijkhout/istc/istc.html (accessed on 16 April 2021).
28. Eijkhout, V. Parallel Programming in MPI and OpenMP; 2016. Available online: https://pages.tacc.utexas.edu/$\sim$eijkhout/pcse/html/index.html (accessed on 16 April 2021).
29. Czarnul, P.; Proficz, J.; Krzywaniak, A. Energy-Aware High-Performance Computing: Survey of State-of-the-Art Tools, Techniques, and Environments. *Sci. Program.* **2019**, *2019*, 8348791. [CrossRef]
30. Krzywaniak, A.; Proficz, J.; Czarnul, P. Analyzing Energy/Performance Trade-Offs with Power Capping for Parallel Applications On Modern Multi and Many Core Processors. In Proceedings of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS), Poznań, Poland, 9–12 September 2018; pp. 339–346.
31. Krzywaniak, A.; Czarnul, P.; Proficz, J. Extended Investigation of Performance-Energy Trade-Offs under Power Capping in HPC Environments; 2019. In Proceedings of the High Performance Computing Systems Conference, International Workshop on Optimization Issues in Energy Efficient HPC & Distributed Systems, Dublin, Ireland, 15–19 July 2019.
32. Prabhakar, A.; Getov, V.; Chapman, B. Performance Comparisons of Basic OpenMP Constructs. In *High Performance Computing*; Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 413–424.
33. Berrendorf, R.; Nieken, G. Performance characteristics for OpenMP constructs on different parallel computer architectures. *Concurr. Pract. Exp.* **2000**, *12*, 1261–1273. [CrossRef]

MOST WIEDZY Downloaded from mostwiedzy.pl