Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management

DOI:
10.1145/2967938.2967946

Document Version
Accepted author manuscript

Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management

Andi Drebes
The University of Manchester
School of Computer Science
Oxford Road
Manchester M13 9PL
United Kingdom
andi.drebes@manchester.ac.uk

Antoniu Pop
The University of Manchester
School of Computer Science
Oxford Road
Manchester M13 9PL
United Kingdom
antoniu.pop@manchester.ac.uk

Karine Heydemann
Sorbonne Universités
UPMC Univ Paris 06
CNRS, UMR 7606, LIP6
4, Place Jussieu
F-75252 Paris Cedex 05
France
karine.heydemann@lip6.fr

Albert Cohen
INRIA and École Normale Supérieure
45 rue d'Ulm
F-75005 Paris
France
albert.cohen@inria.fr

Nathalie Drach
Sorbonne Universités
UPMC Univ Paris 06
CNRS, UMR 7606, LIP6
4, Place Jussieu
F-75252 Paris Cedex 05
France
nathalie.drach-temam@upmc.fr

ABSTRACT

Dynamic task-parallel programming models are popular on shared-memory systems, promising enhanced scalability, load balancing and locality. These promises, however, are undermined by non-uniform memory access (NUMA). We show that using NUMA-aware task and data placement, it is possible to preserve the uniform hardware abstraction of contemporary task-parallel programming models for both computing and memory resources with high data locality. Our data placement scheme guarantees that all accesses to task output data target the local memory of the accessing core. The complementary task placement heuristic improves the locality of accesses to task input data on a best effort basis. Our algorithms take advantage of data-flow style task parallelism, where the privatization of task data enhances scalability by eliminating false dependences and enabling fine-grained dynamic control over data placement. The algorithms are fully automatic, application-independent, performance-portable across NUMA machines, and adapt to dynamic changes. Placement decisions use information about inter-task data dependences readily available in the run-time system, and placement information from the operating system. On a 192-core system with 24 NUMA nodes, our optimizations achieve above 94% locality (fraction of local memory accesses), up to 5× better performance than NUMA-aware hierarchical work-stealing, and even 5.6× compared to static interleaved allocation. Finally, we show that state-of-the-art dynamic page migration by the operating system cannot catch up with frequent affinity changes between cores and data and thus fails to accelerate task-parallel applications.

1. INTRODUCTION

High-performance systems are composed of hundreds of general-purpose computing units and dozens of memory controllers to satisfy the ever-increasing need for computing power and memory bandwidth. Shared memory programming models with fine-grained concurrency have successfully harnessed the computational resources of such architectures [3, 33, 28, 30, 31, 10, 19, 8, 9, 7, 35]. In these models, parallelism is exposed by the programmer through the creation of fine-grained units of work, called tasks, and the specification of synchronization that constrains the order of their execution. A run-time system manages execution of the task-parallel application and acts as an abstraction layer between the program and the underlying hardware and software environment. That is, the run-time is responsible for bookkeeping activities necessary for the correctness of the execution (e.g., the creation and destruction of tasks and their synchronization), interfacing with the operating system for resource management (e.g., allocation of data and meta-data for tasks, scheduling tasks to cores) and efficient exploitation of the hardware.

This concept relieves the programmer from dealing with details of the target platform and thus greatly improves productivity. Yet it leaves issues related to efficient interaction with system software, efficient exploitation of the hardware, and performance portability to the run-time. On today's systems with non-uniform memory access (NUMA), in which memory latency depends on the distance between the requesting cores and the targeted memory controllers, efficient resource usage through task scheduling needs to go hand in hand with the optimization of memory accesses through the placement of physical pages. That is, memory accesses must be kept local in order to reduce latency and data must be distributed across memory controllers to avoid contention.

The alternative of abstracting computing resources only and leaving NUMA-specific optimization to the application is far less attractive. The programmer would have to take into account the different characteristics of all target systems (e.g., the number of NUMA nodes, their associated amount of memory and access latencies), to partition application data properly and to place the data explicitly using operating system-specific interfaces. For applications with dynamic, data-dependent behavior, the programmer would also have to provide mechanisms that constantly react to changes throughout the execution as an initial placement with high data locality at the beginning might have to be revised later on. Such changes would have to be coordinated with the run-time system to prevent destructive performance interference, introducing a tight and undesired coupling between the run-time and the application.

On the operating system side, optimizations are compelled to place tasks and data conservatively [13, 24], unless provided with detailed affinity information by the application [5, 6], high-level libraries [26] or domain specific languages [20]. Nevertheless, as task-parallel run-times operate in user-space, a separate kernel component would add additional complexity to the solution; this advocates for a user-space approach.

This paper shows that it is possible to efficiently and portably exploit dynamic task parallelism on NUMA machines without exposing programmers to the complexity of these systems, preserving a simple, uniform abstract view for both memory and computations, yet achieving high locality of memory accesses. Our solution exploits the "task-private" or data-flow programming style of advanced task-parallel frameworks, allowing the run-time to determine a task's working set and enabling transparent, fine-grained control over task and data placement.

Based on the properties of task-private data, we propose a dynamic task and data placement algorithm to ensure that input and output data are local, and that interacts constructively with work-stealing to provide load-balancing both across cores and memory controllers:

- Our mechanism for memory allocation, called *deferred allocation*, avoids making early placement decisions that could later harm performance. In particular, the memory to store task output data is not allocated until the task placement is known. The mechanism hands over this responsibility to the producer task, on its local NUMA node. *This scheme provides a guarantee for local accesses to all task output data.* Control over data placement is made possible through the *privatization* of task output data.

- To enhance the locality of read memory accesses, we propose *enhanced work-pushing*, a work-sharing mechanism building on earlier work [14] and that interacts constructively with deferred allocation. Since the inputs of a task are outputs of another task, the location of input data is determined by deferred allocation when the producer tasks execute. Enhanced work-pushing is a best-effort mechanism that places a task according to these locations before task execution and thus *before* allocating memory for the task's outputs.

This combination of *enhanced work-pushing* and *deferred allocation* is fully automatic, application-independent, portable across NUMA machines and transparently adapts to dynamic changes at run time. The detailed information about the affinities between tasks and data required by these techniques is either readily available or can be obtained automatically in the run-times of recent task-parallel programming models, such as StarSs [30], OpenMP 4 [28], SWAN [33] and OpenStream [31], which allow the programmer to make inter-task data dependences explicit. While specifying the precise task-level data-flow rather than synchronization constraints alone requires more initial work for programmers, this effort is more than offset by the resulting benefits in terms of performance and performance portability.

The paper is organized as follows. Section 2 presents the principles of enhanced work-pushing. For a more complete discussion of our solutions, we propose multiple heuristics taking into account the placement of input data, output data or both. Section 3 introduces deferred allocation, including a brief outline of the technical solutions employed for fine-grained data placement. Sections 4 and 5 present the experimental methodology and results. A comparison with dynamic page migration is presented in Section 6. Section 7 discusses the most closely related work, before we conclude in Section 8.

2. TASK SCHEDULING WITH ENHANCED WORK-PUSHING

Let us start with terminology and hypotheses about the programming and execution models.

2.1 An abstract model for task parallelism

Our solutions are based on shared memory task-parallel programming models with data dependences. Each task is associated with a set of *incoming data dependences* and a set of *outgoing data dependences*, as illustrated by Figure 1. Each dependence is associated with a contiguous region of memory, called *input buffer* and *output buffer* for incoming and outgoing dependences, respectively. The addresses of these buffers are collected in the task's *frame*, akin to the activation frame storing a function's arguments and local variables in the call stack. While the frame is unique and allocated at the task's creation time, its input and output buffers may be placed on different NUMA nodes and allocated later in the life cycle of the task, but no later than the beginning of the execution of the task, reading from input buffers and writing into output buffers. Buffer placement and allocation time have a direct influence on locality and task-data affinity. Since we ought to offer a uniform abstraction of NUMA resources, we assume input and output buffers are managed by the run-time system rather than explicitly allocated by the application. This is the case of programming models such as OpenStream [31] and CnC [8], but not of StarSs [30] and OpenMP 4.0; see Section 7 for further discussion. We say that a task t_c *depends* on another task t_p if t_p has an outgoing dependence associated to a buffer b and if t_c has an incoming dependence associated to b. In this scenario t_p is referred to as the *producer* and t_c is the *consumer*.

Although this is not a fundamental requirement in our work, we will assume for simplicity that a task executes from beginning to end without interruption. A task becomes *ready* for execution when all of its dependences have been satisfied, i.e., when its producers of input data have

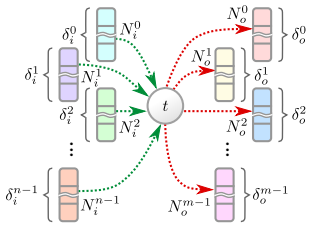


Figure 1: Most general case for a task t: n inputs of size $\delta_i^0, \ldots, \delta_i^{n-1}$, m outputs of size $\delta_o^0, \ldots, \delta_o^{m-1}$, data placed on $n+m$ NUMA nodes $N_i^0, \ldots, N_i^{n-1}, N_o^0, \ldots, N_o^{m-1}$.

completed *and* when its consumers have been created with the addresses of their frames communicated to the task. The *working set* of a task is defined as the union of all memory addresses that are accessed during its execution. Note that the working set does not have to be identical with the union of a task's input and output buffers (e.g., if a task accesses globally shared data structures). However, since our algorithms require accurate communication volume information, we assume that the bulk of the working set of each task is constituted by its input and output data. This is the case for all of the benchmarks studied in the experimental evaluation.

A *worker* thread is responsible for the execution of tasks on its associated core. Each worker is provided with a queue of tasks ready for execution from which it pops and executes tasks. When the queue is empty, the worker may obtain a task from another one through *work-stealing* [4]. A task is pushed to the queue by the worker that satisfies its last remaining dependence. We say that this worker *activates* the task and becomes its *owner*.

The execution of a task-parallel program starts with a *root task* derived from the main function of a sequential process. New tasks are created dynamically. The part of the program involved in creating new tasks is called the *control program*. If only the root task creates other tasks we speak of a *sequential control program*, otherwise of a *parallel control program*.

2.2 Weaknesses of task parallelism on NUMA systems

Whether memory accesses during the execution of a task target the local memory controller of the executing core or some remote memory controller depends on the placement of the input and output buffers and on the worker executing the task. These affinities are highly dynamic and can depend on many factors, such as:

- the order of task creations by the control program;
- the execution order of producers;
- the duration of each task;
- work-stealing events;
- or resource availability (e.g., available memory per node).

In earlier work [14], we showed that some of these issues can be mitigated by using *work-pushing*. Similar to the abstract model discussed above, the approach assumes that tasks communicate through task-private buffers. However, it also assumes that all input data of a task is stored in a single, contiguous memory region rather than multiple input buffers. As a consequence, the task's input data is entirely located on a single node. This property is used by the work-pushing technique, in which the worker activating a task only becomes its owner if the task's input data is stored on the worker's node. If the input data is located on another node, the task is transferred to a random worker associated to a core on that node. The approach remains limited however: (1) as it assumes that all inputs are located on the same node it is ill-suited for input data located on multiple nodes, and (2) it does not optimize for outgoing dependences.

Below, we first present *enhanced work-pushing*, a generalization of work-pushing, capable of dealing with input data distributed over multiple input buffers potentially placed on different nodes. This technique serves as a basis for the complementary *deferred allocation* technique, presented in the next section, that allows the run-time to improve the placement of output buffers. We introduce three *work-pushing heuristics* that schedule a task according to the placement of its input or output data or both. This complements the study of the effects of work-pushing on data locality and experimentally underpins the limitations of NUMA-aware scheduling alone, limited to passive reactions to a given data placement in Section 5.

2.3 Enhanced work-pushing

The names of the three heuristics for enhanced work-pushing are *input-only*, *output-only* and *weighted*. The first two heuristics take into account incoming and outgoing dependences only, respectively. The *weighted* heuristic takes into account all dependences, but associates different weights to incoming and outgoing dependences to honor the fact that read and write accesses usually do not have the same latency.

Algorithm 1 shows how the heuristics above are used by the function *activate*, which is called when a worker w activates a task t (i.e., when the task becomes ready). Lines 1 to 3 define variables u_{in} and u_{out}, indicating which types of dependences should be taken into account according to the heuristic h. The variables used to determine whether the newly activated task needs to be transferred to a remote node are initialized in lines 5 to 8: the *data* array stores the cumulated size of input and output buffers of t for each of the N nodes of the system, D_{in} stands for the incoming dependences of t and D_{out} for its outgoing dependences.

The for loop starting in Line 10 iterates over a list of triples with a set of dependences D_{cur}, a variable u_{cur} indicating whether the set should be taken into account, and a weight w_{cur} associated to each type of dependence. During the first iteration, D_{cur} is identical with D_{in} and during the second iteration identical with D_{out}. For each dependence in D_{cur} a second loop in Line 12 determines the buffer b used by the dependence, the size s_{b} of the buffer as well as the node n_{b} containing the buffer. The node on which a buffer is placed might be unknown if the buffer has not yet been placed by the operating system, e.g., if it has been allocated using the first-touch mechanism and has not yet been written. If this is the case, its size is added to the total size s_{tot}, but not included into the per-node statistics. Otherwise, the *data* array is updated accordingly by multiplying s_{b} with the weight w_{cur}.

Once the total size and the weighted number of bytes per node have been determined, the procedure checks whether the task should be pushed to a remote node. Tasks whose overall size of dependences is below a threshold are added to the local queue (Line 24) to avoid cases in which the

Algorithm 1: activate(w, t)

```
1  if h = input only then (u_in, u_out) ← (true, false)
2  else if h = output only then (u_in, u_out) ← (false, true)
3  else if h = weighted then (u_in, u_out) ← (true, true)
4
5  data[0, ... N − 1] ← ⟨0, ..., 0⟩
6  D_in ← in_deps(t)
7  D_out ← out_deps(t)
8  s_tot ← 0
9
10 for (D_cur, u_cur, w_cur) in
   ⟨(D_in, u_in, w_in), (D_out, u_out, w_out)⟩ do
11    if u_cur = true then
12       for d ∈ D_cur do
13          s_b ← size_of(buffer_of(d))
14          n_b ← node_of(buffer_of(d))
15          s_tot ← s_tot + s_b
16          if n_b ≠ unknown then
17             data[n_b] ← data[n_b] + w_cur · s_b
18          end
19       end
20    end
21 end
22
23 if s_tot < threshold then
24    add_to_local_queue(w, t)
25 else
26    n_min ← node_with_min_access_cost(data)
27
28    if n_min ≠ local_node_of_worker(w) then
29       w_dst ← random_worker_on_node(n_min)
30       res ← transfer_task(t, w_dst)
31       if res = failure then
32          add_to_local_queue(w, t)
33       end
34    else
35       add_to_local_queue(w, t)
36    end
37 end
```

overhead of a remote push cannot be compensated by the improvement on execution time. For tasks with larger dependences, the run-time determines the node $n_{\min}$ with the minimal overall access cost (Line 29). The access cost for a node N_i is estimated by summing up the access costs to each node N_j containing at least one of the buffers, which in turn can be estimated by multiplying the average latency between N_i and N_j.[1] If $n_{\min}$ is different from the local node n_{lcl} of the activating worker, the run-time tries to transfer t to a random worker on $n_{\min}$. If this fails, e.g., if the data structure of the targeted worker receiving remotely pushed tasks is full, the task is added to the local queue (Line 32).

2.4 Limitations of enhanced work-pushing

The major limitation of enhanced work-pushing is that, regardless of the heuristic, it can only react passively to a given data placement. This implies that data must already be well-distributed across memory controllers if all tasks should take advantage of this scheduling strategy. For poorly distributed data, e.g., if all data is placed on a single node, a subset of the workers receives a significantly higher amount of tasks than others. Work-stealing redistributes tasks among the remaining workers and thus prevents the system from load imbalance, but cannot improve overall data locality if the initial data distribution was poor. Classical task parallel run-times allocate buffers during task creation [8, 33, 31]; hence data distribution mainly depends on the control program. A sequential control program leads to poorly placed data, while a parallel control program lets work-stealing evenly distribute task creation and buffer allocation. However, writing a parallel control program is already challenging in itself, even for programs with regularly-structured task graphs. Additionally ensuring an equal distribution of data across NUMA nodes through the control program is even more challenging or infeasible, especially for applications with less regularly-structured task graphs (e.g., if the structure of the graph depends on input data). Such optimizations also reject efficient exploitation of NUMA to the programmer and are thus contrary to the idea of abstraction from the hardware by the run-time.

In the following section, we introduce a NUMA-aware allocator that complements the *input only* work-pushing heuristic and that decouples data locality from the control program, leaving efficient exploitation to the run-time.

[1]We estimated the latencies based on the distance between each pair of nodes reported by the NUMACTL tool provided by LIBNUMA [21].

3. DEFERRED ALLOCATION

NUMA-aware allocation controls the placement of data on specific nodes. Our proposed scheme to make these decisions transparent relies on *per-node memory pools* to control the placement of task buffers.

3.1 Per-node memory pools

Per-node memory pools combine a mechanism for efficient reuse of blocks of memory with the ability to determine on which nodes blocks are placed. Each NUMA node has a memory pool that is composed of k free lists L_0 to L_{k-1}, where L_i contains blocks of size $2^{S_{\min}+i}$ bytes. When a worker allocates a block of size s, it determines the corresponding list L_j with $2^{S_{\min}+j-1} < s \leq 2^{S_{\min}+j}$ and removes the first block of that list. If the list is empty, it allocates a larger chunk of memory from the operating system, removes the first block from the chunk and adds the remaining parts to the free list.

A common allocation strategy of operating systems is first-touch allocation, composed of two steps. The first step referred to as *logical allocation* is triggered by the system call used by the application to request additional memory and only extends the application's address space. The actual *physical allocation* is triggered upon the first write to the memory region and places the corresponding page on the same node as the writing core. Hence, a block that originates from a newly allocated chunk is not necessarily placed on any node.

However, when a block is freed, it has been written by a producer and it is thus safe to assume that the block has been placed through physical allocation. The identifier of the containing node can be obtained through a system call, which enables the run-time to return the block to the correct memory pool. To avoid the overhead of a system call each time a block is freed, information on the NUMA node containing a block is cached in a small meta-data section associated to the block. This memory pooling mechanism provides three fundamental properties for deferred allocation presented below. First, it ensures that allocating a block from a memory pool always returns a block that has not

been placed yet or a block that is known to be placed on the node associated to the memory pool. Second, data can be placed on a specific node with very low overhead. Finally, the granularity for data placement is decoupled from the usual page granularity as a block may be sized arbitrarily.

3.2 Principles of deferred allocation

The key idea of deferred allocation is to delay the allocation and thus the placement of each task buffer until the node executing the producer that writes to the buffer is known in order to guarantee that accesses to output buffers are always local. The classical approach in run-times for dependent tasks is to allocate input buffers upon the creation of a task [8, 33, 31] or earlier [30]. Instead, we propose to let each input buffer for a consumer task t_c to be allocated by the producer task t_p writing into it, immediately before task t_c starts execution. Since the input buffer of t_c is an output buffer of t_p, the location of input data in t_c is effectively determined by its producer(s). In the following, we use the term *immediate allocation* to distinguish the default allocation scheme in which input buffers are allocated upon creation from deferred allocation.

Figure 2a shows the implications of immediate allocation on data locality for a task t. All input buffers of t are allocated on the node N_c on which the creator of t operates. The same scheme applies to the creators $t_{c,o}^0$ to $t_{c,o}^{m-1}$, causing the input buffers of the tasks t_o^0 to t_o^{m-1} to be allocated on nodes N_o^0 to N_o^{m-1}, respectively. In the worst case for data locality, t is stolen by a worker operating on neither N_c nor N_o^0 to N_o^{m-1} and all memory accesses of t target memory on remote nodes.

When using deferred allocation, the input buffers of t are not allocated before its producers start execution and the output buffers of t are not allocated before t is activated (Figure 2b). When t becomes ready, all of its input buffers have received input data from the producers of t and have been placed on up to n different nodes N_i^0 to N_i^{n-1} (Figure 2c). The data locality impact of deferred allocation is illustrated in Figure 2d, showing the placement at the moment when the worker executing t has been determined. Regardless of any possible placement of t, all of its output buffers are placed on the same node as the worker executing the task. Hence, using deferred allocation, write accesses are guaranteed to target local memory. Furthermore, this property is independent from the placement of the creating tasks t_c and $t_{c,o}^0$ to $t_{c,o}^{m-1}$, which effectively decouples data locality from the control program. Even for a sequential control program data is distributed over the different nodes of the machine according to work-stealing. This way, work-stealing does not only take the role of a mechanism responsible for computational load balancing, but also the role as a mechanism for load balancing across memory controllers.

An important side effect of deferred allocation is a significant reduction of the memory footprint. With a sequential control program, all tasks are created by a single "root" task. This causes a large number of input buffers to be allocated early on, while the actual working set of live buffers might be much smaller. A parallel control program can mitigate the effects of early allocation, e.g., by manually throttling task creation as shown in Figure 3b. However, this requires significant programmer effort and hurts the separation of concerns that led to the delegation of task management to the run-time.

Thanks to deferred allocation, buffers allocated for early tasks can be reused at a later stage. The difference is shown in Figures 3a and 3c. In the first case, all three buffers b_i, b_{i+1} and b_{i+2} are allocated before the dependent tasks t_i to t_{i+3} are executed. In the latter case, the buffer used by t_i and t_{i+1} can be reused as the input buffer of t_{i+3}. Parallel control programs also benefit from deferred allocation as the minimal number of buffers along a path of dependent tasks can be decreased by one (e.g., in Figure 3b only b_i and b_{i+1} are simultaneously live and b_i can be reused for b_{i+2} when using deferred allocation).

3.3 Compatibility with work-pushing

Deferred allocation guarantees local write accesses, but it does not influence the locality of read accesses. By combining deferred allocation with the input-only heuristic of enhanced work-pushing, it is possible to optimize for both read and write accesses.

It is important to note that neither the output-only heuristic nor the weighted heuristic can be used since the output buffers of a task are not determined upon task activation when the work-pushing decision is taken.

4. EXPERIMENTAL SETUP

For the experimental evaluation we implemented enhanced work-pushing and deferred allocation in the run-time system of the OpenStream project [29]. We start with an overview of the software and hardware environment used in our experiments, followed by a presentation of the selected benchmarks.

4.1 Software environment

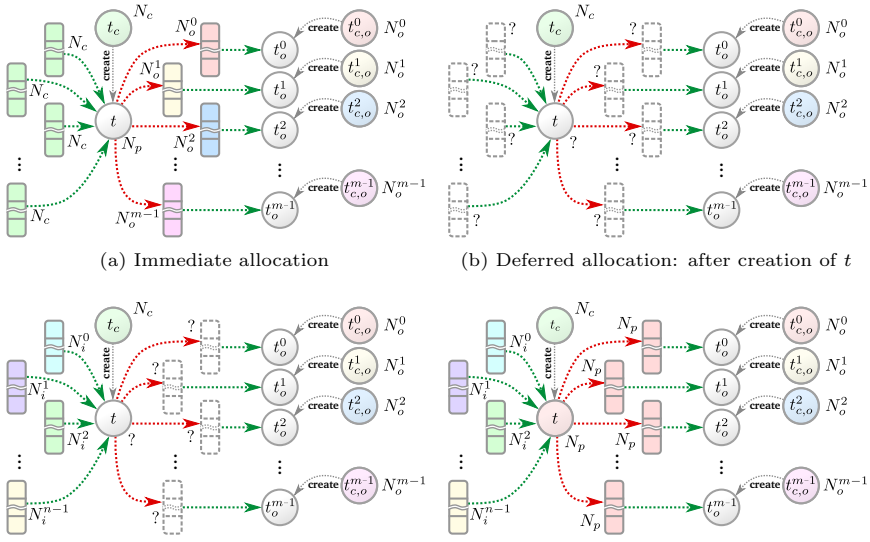
OpenStream [31] is a task-parallel, data-flow programming model implemented as an extension to OpenMP. Arbitrary dependence patterns can be used to exploit task, pipeline and data parallelism. Each data-flow dependence is semantically equivalent to a communication and synchronization event within an unbounded FIFO queue referred to as a stream. Pragmatically, this is implemented by compiling dependences as accesses to task buffers dynamically allocated at execution time: writes to streams result in writes to the buffers of the tasks consuming the data, while read accesses to streams by consumer tasks are translated to reads from their own, task-private buffers.

We implemented the optimizations presented in this paper into the publicly available run-time of OpenStream [29]. Crucially, we rely on the fact that OpenStream programs are written with programmer annotations explicitly describing the flow of data between tasks. This precise data-flow information is preserved during compilation and made available to the run-time library. We leverage this essential semantic information to determine, at run-time, and *before* task execution, how much data is exchanged by any given task.

OpenStream programs are dynamically load-balanced, worker threads use hierarchical work-stealing to acquire and execute tasks whose dependences have been satisfied. If work-pushing is enabled, workers can also receive tasks in a dedicated multi-producer single-consumer queue [14]. Our experiments use one worker thread per core.

4.2 Hardware environment

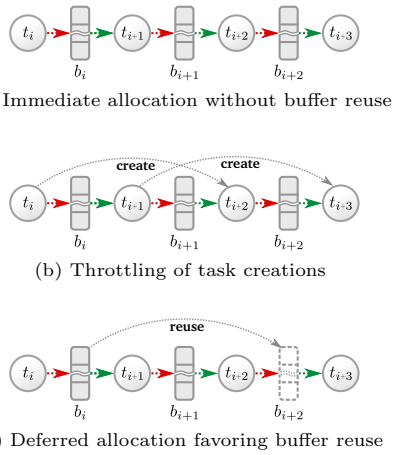
The experiments were conducted on two many-core systems.



(a) Immediate allocation (b) Deferred allocation: after creation of t

(c) Deferred allocation: activation of t (d) Deferred allocation: final placement

Figure 2: Allocation schemes



(a) Immediate allocation without buffer reuse

(b) Throttling of task creations

(c) Deferred allocation favoring buffer reuse

Figure 3: Allocation and reuse

Opteron-64 is a quad-socket system with four AMD Opteron 6282 SE processors running at 2.6GHz, using Scientific Linux 6.2 with kernel 3.10.1. The machine is composed of 4 physical packages, with 2 dies per package, each die containing 8 cores organized in pairs. Each pair shares the first-level instruction cache as well as a 2MiB L2 cache. An L3 cache of 6MiB and the memory controller are shared by the 8 cores on the same die. The 16KiB L1 cache is private to each core. Main memory is 64GiB, equally divided into 8 NUMA domains. For each NUMA node, 4 neighbors are at a distance of 1 hop and 3 neighbors are at 2 hops.

SGI-192 is an SGI UV2000 with 192 cores and 756GiB RAM, distributed over 24 NUMA nodes, and running SUSE Linux Enterprise Server 11 SP3 with kernel 3.0.101-0.46-default. The system is organized in *blades*, each of which contains two Intel Xeon E5-4640 CPUs running at 2.4GHz. Each CPU has 8 cores with direct access to a memory controller. The cache hierarchy consists of 3 levels: a core-private L1 with separate instruction and data cache, each with a capacity of 32KiB; a core-private, unified L2 cache of 256KiB; and a unified L3 cache of 20MiB, shared among all 8 cores of the CPU. *Hyperthreading* was disabled for our experiments. Each blade has a direct connection to a set of other blades and indirect connections to the remaining ones. From a core's perspective, a memory controller can be either local if associated to the same CPU, at 1 hop if on the same blade, at 2 hops if on a different blade that is connected directly to the core's blade or at 3 hops if on a remote blade with an indirect connection.

Latency of memory accesses and NUMA factors.

We used a synthetic benchmark to measure the latency of memory accesses as a function of the distance in hops between a requesting core and the memory controller that satisfies the request. It allocates a buffer on a given node using LibNUMA, initializes it and measures execution time for a sequence of memory accesses to this buffer from a core on a specific node. Each sequence traverses the whole buffer from beginning to end in steps of 64 bytes, such that each cache line is only accessed once. The buffer size was set to 1GiB to ensure data is evicted from the cache before it is reused and thus to measure only memory accesses that are satisfied by the memory controller and not by the hierarchy of caches.

	Read accesses	Write accesses	Factor R/W
Local	1288.5 ± 1.22ms	2256.9 ± 14.06ms	1.00 / 1.00
1 hop (on-package)	2328.4 ± 0.49ms	2717.6 ± 12.16ms	1.81 / 1.20
1 hop (off-package)	2781.4 ± 0.56ms	3934.6 ± 00.56ms	2.16 / 1.74
2 hops	5601.6 ± 0.57ms	5601.3 ± 00.55ms	4.34 / 2.48

Table 1: Average latency of accesses on Opteron-64

	Read accesses	Write accesses	Factor R/W
Local	934.82 ± 5.74ms	1307.4 ± 2.95ms	1.00 / 1.00
1 hop	4563.1 ± 3.02ms	5282.38 ± 1.56ms	4.88 / 4.04
2 hops	5820.48 ± 2.11ms	6473.38 ± 1.16ms	6.23 / 4.95
3 hops	6991.24 ± 2.71ms	7673.14 ± 0.92ms	7.48 / 5.87

Table 2: Average latency of accesses on SGI-192

Tables 1 and 2 indicate the total execution time as a function of the number of hops and the access mode of the synthetic benchmark for both systems. The results show that latency increases with the distance between the requesting core and the targeted memory controller and that writes are significantly slower than reads. The rightmost column of each table shows the access time normalized to accesses targeting local memory. For reads on the Opteron-64 system, these values range from 1.81 for on-package accesses to a memory controller at a distance of one hop to a factor of 4.34 for off-package accesses at a distance of two hops. For writes, these values are lower—1.2 to 2.48—due to the higher baseline latency of local writes. Not surprisingly, the factors for both reads and writes are significantly higher on the larger SGI-192 system (up to 7.48 for reads at three hops). This suggests that locality optimizations will have a higher impact on SGI-192 and that the locality of writes will have the greatest impact.

4.3 Benchmarks

We evaluate the impact of our techniques on nine benchmarks, each of which is available in an optimized sequential implementation and two tuned parallel implementations using OpenStream.

The first parallel implementation uses task-private input and output buffers as described in Section 2.1 and thus enables enhanced work-pushing and deferred allocation. Data from input buffers is only read and never written, while data in output buffers is only written and never read. Hence, tasks cannot perform in-place updates and results are written to a different location than the input data. We refer to this implementation as *DSA* (dynamic single assignment).

The second parallel implementation, which we refer to as *SHM*, uses globally shared data structures and thus does not expose information on memory accesses to the run-time. However, the pages of the data structures are distributed across all NUMA nodes in a round-robin fashion using *interleaved allocation*. We use this implementation to compare our solutions to classical static NUMA-aware optimizations that require only minimal changes to the application. The benchmarks are the following.

- *Jacobi-1d*, *jacobi-2d* and *jacobi-3d* are the usual one-, two- and three-dimensional Jacobi stencils iterating over arrays of double precision floating point elements. At each iteration, the algorithm averages for each matrix element the values of the elements in its Von Neumann neighborhood using the values from the previous iteration.
- *Seidel-1d*, *seidel-2d* and *seidel-3d* employ a similar stencil pattern but use values from the previous and the current iteration for updates.
- *Kmeans* is a data-mining benchmark that partitions a set of n d-dimensional points into k clusters using the K-means clustering algorithm. Each vector is represented by d single precision floating point values.
- *Blur-roberts* applies two successive image filters on double precision floating point elements [22]: a Gaussian *blur filter* on each pixel's Moore neighborhood followed by the *Roberts Cross Operator* for edge detection.
- *Bitonic* implements a bitonic sorting network [1], applied to a sequence of arbitrary 64-bit integers.

Table 3 summarizes the parameters for the different benchmarks and machines. The size of input data was chosen to be significantly higher than the total amount of cache memory and low enough to prevent the system from swapping. This size is identical on both machines, except for *blur-roberts*, whose execution time for images of size $2^{15}\times2^{15}$ is too short on SGI-192 and which starts swapping for size $2^{16}\times2^{16}$ on Opteron-64. To amortize the execution of auxiliary tasks at the beginning and the end of execution of the stencils, we set the number of iterations to 60. The block size has been tuned to minimize the execution time for the parallel implementation with task-private input and output data (DSA) on each machine. To avoid any bias in favor of our optimizations, enhanced work-pushing and deferred allocation have been disabled during this tuning phase. In this configuration, the run-time only relies on optimized *work-stealing* [23] extended with *hierarchical work-stealing* [14] for computational load balancing. We refer to this *baseline* for our experiments as *DSA-BASE*. Identical parameters for the block size and run-time have been used for the experiments with the shared memory versions of the benchmarks (SHM), which we refer to as *SHM-BASE*.

	Matrix / Vector size	**Block size** (Opteron-64)	**Block size** (SGI-192)	**Iterations**
Jacobi-1d	2^{28}	2^{16}	2^{16}	60
Jacobi-2d	$2^{14}\times2^{14}$	$2^{10}\times2^{6}$	$2^{8}\times2^{8}$	60
Jacobi-3d	$2^{10}\times2^{9}\times2^{9}$	$2^{5}\times2^{6}\times2^{5}$	$2^{4}\times2^{6}\times2^{6}$	60
Seidel-1d	2^{28}	2^{16}	2^{16}	60
Seidel-2d	$2^{14}\times2^{14}$	$2^{8}\times2^{8}$	$2^{7}\times2^{9}$	60
Seidel-3d	$2^{10}\times2^{9}\times2^{9}$	$2^{6}\times2^{5}\times2^{5}$	$2^{4}\times2^{8}\times2^{4}$	60
Blur-roberts	$2^{15}\times2^{15}$ (Opteron-64) $2^{16}\times2^{16}$ (SGI-192)	$2^{9}\times2^{6}$	$2^{10}\times2^{6}$	-
Bitonic	2^{28}	2^{16}	2^{17}	-
K-means	40.96M pts, 10 dims., 11 clust.	10^4	10^4	-

Table 3: Benchmark parameters

All benchmarks were compiled using the OpenStream compiler based on GCC 4.7.0. The compilation flags for *blur-roberts* as well as the *jacobi* and *seidel* benchmarks were `-O3 -ffast-math`, while *kmeans* uses `-O3` and *bitonic* uses `-O2`.

The parallel implementations are provided with a parallel control program to prevent sequential task creation from becoming a performance bottleneck. To avoid memory controller contention, the initial and final data are stored in global data structures allocated using interleaved allocation across all NUMA nodes.

Data dependence patterns.

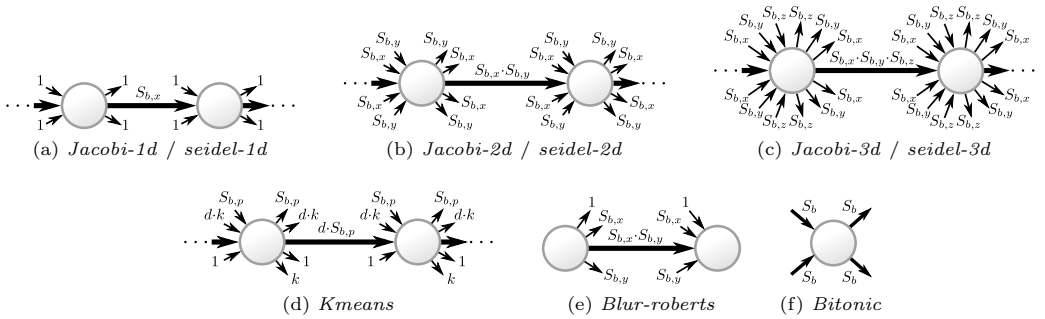
The relevant producer-consumer patterns shown in Figure 4 can be divided into three groups with different implications for our optimizations: *unbalanced dependences* (e.g., one input buffer accounting for more than 90% of the input data) with long dependence paths (*jacobi-1d*, *jacobi-2d*, *jacobi-3d*, *seidel-1d*, *seidel-2d*, *seidel-3d*, *kmeans*), *unbalanced dependences* with short dependence paths (*blur-roberts*) and *balanced dependences* (*bitonic*). The behavior of our heuristics on these patterns is referenced in the experimental evaluation. All of the benchmarks have non-trivial, connected task graphs, i.e., none of the benchmarks represents an embarrassingly parallel workload.

Characterization of memory accesses.

The benchmarks were carefully tuned (block sizes and tiling) to take advantage of caches. However, the efficiency of the cache hierarchy also depends on the pattern, the frequency and the timing of memory accesses during the execution of a benchmark, leading to more or fewer cache misses for a given block size. Figure 5 shows the cache miss rates at the last level of cache (LLC) on SGI-192 of DSA-BASE, which is a good proxy for the rate of requests to main memory for each benchmark. For all bar graphs in this paper, error bars indicate standard deviation. As the focus of our optimizations is on the locality of memory accesses, we expect a higher impact for benchmarks exhibiting higher LLC miss rates. For this reason, *seidel* and *blur-roberts* are expected to benefit the most from our optimizations, followed by the *jacobi* benchmarks and *bitonic*. *Kmeans* has a very low LLC miss rate and is not expected to show significant improvement.

4.4 Experimental baseline

To demonstrate the effectiveness of our optimizations, our



(a) *Jacobi-1d / seidel-1d* (b) *Jacobi-2d / seidel-2d* (c) *Jacobi-3d / seidel-3d*

(d) *Kmeans* (e) *Blur-roberts* (f) *Bitonic*

Figure 4: Main tasks and types of dependences of the benchmarks. The amount of data exchanged between tasks is indicated by the width of arrows. Symbols: $S_{b,x}$, $S_{b,y}$, $S_{b,z}$: number of elements per block in x, y and z direction; $S_{b,p}$: number of points per block, d: number of dimensions, k: number of clusters in *kmeans*; S_{b}: number of elements per block in *bitonic*.

Figure 5: LLC misses per thousand instructions on SGI-192

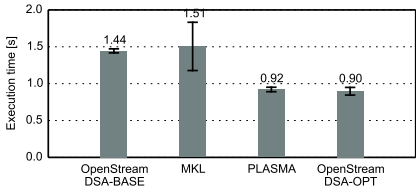


Figure 6: Cholesky factorization on Opteron-64.

principal point of comparison is DSA-BASE. We validate the soundness of this baseline by comparing its performance on Cholesky factorization against the two state-of-the-art linear algebra libraries PLASMA/QUARK [35] and Intel's MKL [18]. Figure 6 shows the execution times of Cholesky factorization on a matrix of size 8192×8192 running on the Opteron-64 platform with four configurations: DSA-BASE, Intel MKL, PLASMA and finally optimized OpenStream (DSA-OPT), the latter using our run-time implementing the optimizations presented in this paper. This validates the soundness of our baseline, which achieves similar performance to Intel MKL, while also showcasing the effectiveness of our optimizations, automatically and transparently matching the performance of PLASMA without any change in the benchmark's source code.

5. RESULTS

We now evaluate enhanced work-pushing and deferred allocation, starting with the impact on memory access locality, and following on with the actual performance results.

5.1 Impact on the locality of memory accesses

On the Opteron platform, we use two hardware performance counters to count the requests to node-local memory[2] and to a remote node[3], respectively. We consider the locality metric $R_{\mathrm{loc}}^{\mathrm{HW}}$, defined as the ratio of local memory accesses to total memory accesses, shown in Figure 7. We could not provide the corresponding figures for the SGI system due to missing support in the kernel. However, the OpenStream run-time contains precise information on the working set of tasks and on the placement of input buffers, which can be used to provide a second locality metric $R_{\mathrm{loc}}^{\mathrm{RT}}$ that precisely accounts for accesses to data managed by the run-time, i.e., associated to task dependences.

Figure 7 shows the locality of requests to main memory $R_{\mathrm{loc}}^{\mathrm{HW}}$ on Opteron-64. Data locality is consistently improved by our optimizations across all benchmarks. The combination of enhanced work-pushing and deferred allocation (DSA-OPT) is comparable to the output only and weighted heuristics of work-pushing, but yields significantly better results than enhanced work-pushing only for benchmarks with balanced dependences. For all *jacobi* and *seidel* benchmarks as well as *kmeans* the locality was improved above 88% and for *bitonic* above 81%. *Blur-roberts* does not benefit as much from our optimizations as the other benchmarks. As the run-time system only manages the placement of privatized data associated with dependences, short dependence paths, such as in *Blur-roberts*, only allow the run-time to optimize placement for a fraction of the execution. As a result, the overall impact is diluted proportionately. We further note that the input only heuristic of work-pushing—closest to the original heuristic in prior work [14]—does not improve memory locality as much as weighted work-pushing or the combination of deferred allocation with work-pushing. Figure 8 shows $R_{\mathrm{loc}}^{\mathrm{RT}}$ on SGI-192, highlighting the effectiveness of our optimizations on data under the control of the run-time. Not accounting for unmanaged data, we achieve almost perfect locality (up to 99.8%) across all benchmarks, except for *bitonic*, where balanced dependences imply that whenever input data is on multiple nodes, only half of the input data can be accessed locally.

5.2 Impact on performance

Figure 9 shows the speedup achieved over DSA-BASE. The best performance is achieved by combining work-pushing and deferred allocation, with a global maximum of 2.5× on Opteron-64 and 5.0× on SGI-192. Generally, the speedups are higher on SGI-192, showing that our optimizations have

[2] *CPU_IO_REQUESTS_TO_MEMORY_IO:LOCAL_CPU_TO_LOCAL_MEM*

[3] *CPU_IO_REQUESTS_TO_MEMORY_IO:LOCAL_CPU_TO_REMOTE_MEM*

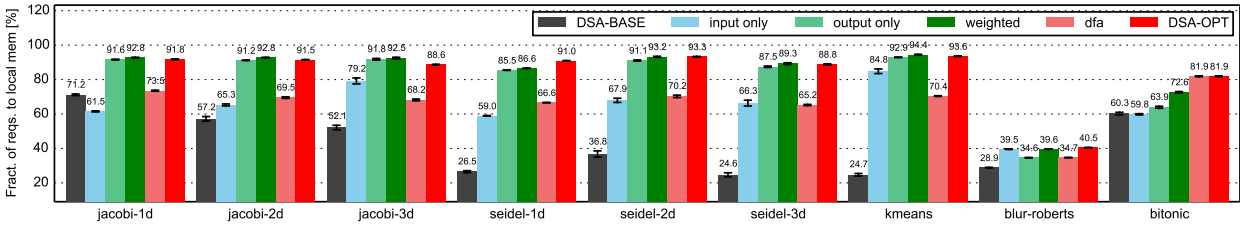


Figure 7: Locality $R_{\text{loc}}^{\text{HW}}$ of requests to main memory on the Opteron-64 system for deferred allocation

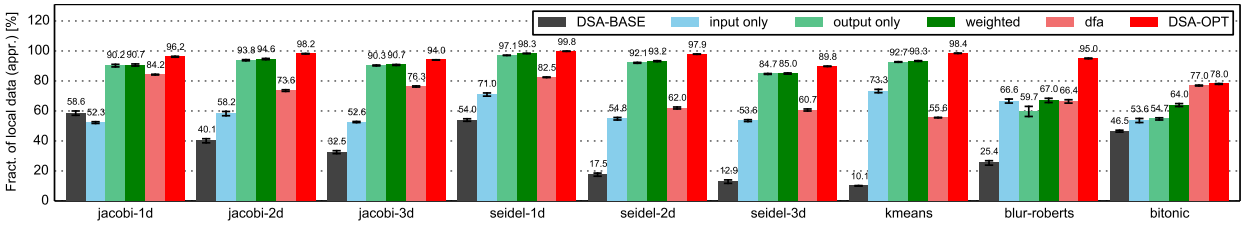


Figure 8: Locality $R_{\text{loc}}^{\text{RT}}$ of data managed by the run-time on SGI-192

a higher impact on machines with higher penalties for remote accesses. These improvements result from better locality as well as from the memory footprint reduction induced by deferred allocation. Note that the input-only heuristic did not perform well with *jacobi*, highlighting the importance of considering both input and output flows, and of proactively distributing buffers through deferred allocation rather than reactively adapting the schedule only.

5.3 Comparison with interleaved allocation

Figure 10 shows the speedup of the parallel baseline over the implementations using globally shared data structures distributed over all NUMA nodes using interleaved allocation (SHM-BASE). The optimizations achieve up to 3.1× speedup on Opteron-64 and 5.6× on SGI-192. The best performance is systematically obtained by the combined work-pushing and deferred allocation strategy. These results clearly indicate that taking advantage of the dynamic data-flow information provided in modern task-dependent languages allows for more precise control over the placement of data leading to improved performance over static schemes unable to react to dynamic behavior at execution time. In the case of interleaved allocation, the uniform access pattern of the benchmarks evaluated in this work yields good load balancing across memory controllers, but poor data locality.

6. COMPARISON WITH DYNAMIC PAGE MIGRATION

While our study focused on the application- and run-time, the reader may wonder how kernel-level optimizations fare in our context. Recent versions of Linux comprise the *balancenuma* patchset [12] for dynamic page migration and a transparent policy migrating pages during the execution of a process based on its memory accesses. The kernel periodically scans the address space of the process and changes the protection flags of the scanned pages such that an access causes an exception. Upon an access to such a page, the exception handler checks whether the page is misplaced with respect to NUMA and migrates it towards the node of the accessing CPU.

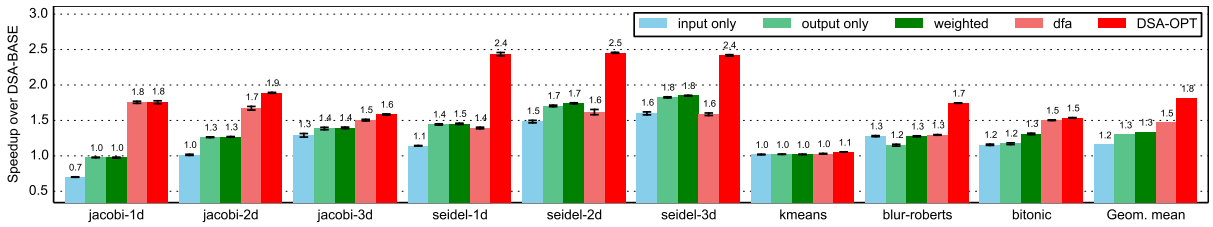
We first evaluate the influence of page migration on a synthetic benchmark to determine under which conditions the mechanism is beneficial, and show that these conditions do not meet the requirements for task-parallel programs. We then study the impact of dynamic page migration on the parallel baseline with globally shared data structures (SHM-BASE).

6.1 Parametrization of page migration

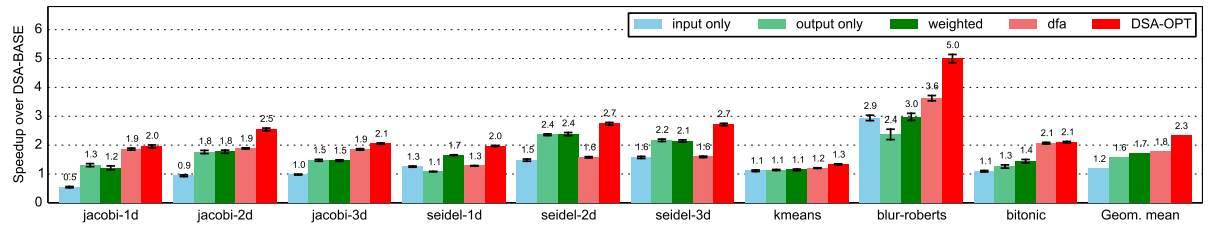
For all experiments, we used version 4.3.0 of the Linux Kernel. As the SGI test platform requires specific kernel patches and is shared among many users, we conducted these experiments on Opteron-64 only. The migration mechanism is configured through the procfs pseudo filesystem, as follows:

- Migration can be globally enabled or disabled by setting *numa_balancing* to 1 or 0, respectively.
- The parameter *numa_balancing_scan_delay_ms* indicates the minimum execution time of a process before page migration starts. In our experiments, we have set this value to 0 to enable migration as soon as possible. Page migration during initialization is prevented using appropriate calls to *mbind*, temporarily imposing static placement.
- The minimum / maximum duration between two scans is controlled by *numa_balancing_scan_period_min_ms* / *numa_balancing_scan_period_max_ms*. We have set the minimal period to 0 and the maximum period to 100000 to allow for constant re-evaluation of the mapping.
- How much of the address space is examined in one scan is defined by *numa_balancing_scan_size_mb*. In the experiments, this parameter has been set to 100000 to prevent the system from scanning only a subset of the pages.

In the following evaluation, we calculate the ratio of the median wall clock execution time with dynamic migration

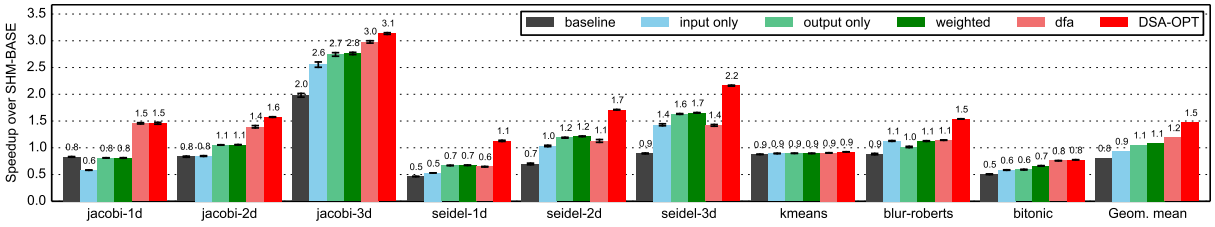


(a) Opteron-64 system

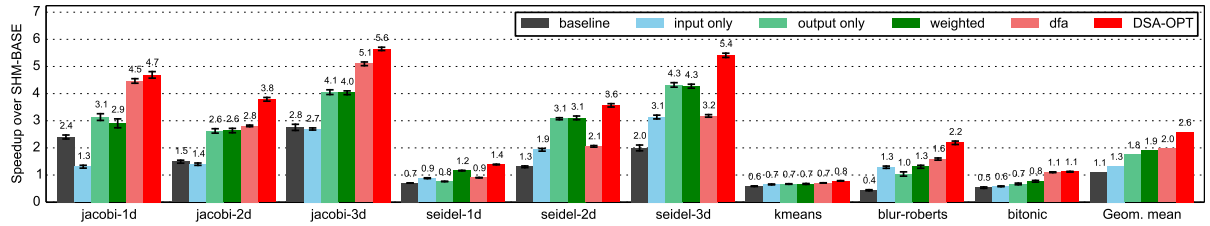


(b) SGI-192 system

Figure 9: Speedup over the parallel baseline DSA-BASE



(a) Opteron-64



(b) SGI-192

Figure 10: Speedup over the implementations with globally shared data structures and interleaving on all nodes (SHM-BASE)

(*numa_balancing* set to 1) divided by the median time without migration (*numa_balancing* set to 0) for 10 runs of a synthetic benchmark.

6.2 Evaluation of a synthetic benchmark

The synthetic benchmark has been designed specifically to evaluate the potential of dynamic page migration for scenarios with clear relationships between data and computations and without interference. It is composed of the following steps:

1. Allocate S sets of T 64MiB buffers, distributed in a round-robin fashion on the machine's NUMA node. That is, the i-th buffer of each set is allocated on NUMA node (i mod N), with N being the total number of NUMA nodes.

2. Create T threads and pin the i-th thread on the i-th core.

3. Assign exactly one buffer of the current set to each thread, with thread i being the owner of the i-th buffer.

4. Synchronize all threads using a barrier and let each thread traverse its buffer I times linearly by adding a constant to the first 8-byte integer of every cache line of the buffer.

5. Change the affinity A times by repeating steps 3 and 4 a total of A times.

6. Synchronize all threads with a barrier and print the time elapsed between the moments in which the first thread passed the first and the last barrier, respectively.

On Opteron-64, the number of 8 cores per NUMA node is equal to the total number of nodes. Thus, the allocation scheme above causes every NUMA node to access every node of the system at the beginning of each affinity change, which

allows for load-balancing on the system's memory controllers and thus factors out contention arising from the initial distribution. We have set the total number of affinity changes A to 8.

Figure 11 shows the speedup for a varying number of iterations I before each affinity change. We found that the preferred page size for the buffers has a strong influence on migration overhead. Therefore, we evaluate three configurations: *default* does not impose any specific page size and leaves the choice to the operating system, while *small* and *huge* force the use of 4KiB and 2MiB pages, respectively.

In order to match the performance of the baseline with static placement, at least 12 iterations are necessary for *default* and *huge*. For small pages, at least 70 iterations must be performed. In our benchmarks with task-private input and output buffers, the bulk of the input data of each task is accessed only up to 7 times. The temporal data locality is thus far below these thresholds, which lets us expect that dynamic page migration cannot improve performance.

6.3 Evaluation of OpenStream benchmarks

Let us now study the impact of page migration on the second parallel baseline with globally shared data structures (SHM-BASE). In contrast to task-private buffers, in which each input buffer at each iteration potentially uses a different set of addresses, each block of data of the globally shared data structures is associated to a fixed set of addresses for all iterations with very high temporal locality. As in the previous experiments, we used hierarchical work-stealing, initial interleaved allocation and the parameters for the benchmarks described in Table 3.

Figure 12 shows the speedup of dynamic page migration over the median execution time without migration. For none of the benchmarks does page migration improve performance. In the best case (*jacobi-3d*, *seidel-2d*, *seidel-3d*, *kmeans*, and *blur-roberts*) performance is almost identical with small variations. In many other cases performance degrades substantially (*jacobi-1d*, *jacobi-2d*, *seidel-1d*, and *bitonic*).

The first reason for this degradation is that dynamic page migration is not able to catch up with frequent affinity changes between cores and data. Second, in contrast to task-private buffers, data blocks of the shared data structures do not necessarily represent contiguous portions of the address space and a single huge page might contain data of more than one block, accessed by different cores. In conclusion, page migration only reacts to changes in the task-data affinity while our scheme proactively binds them together, and page granularity may also not be appropriate for data placement in task-parallel applications.

7. RELATED WORK

Combined NUMA-aware scheduling and data placement can be split into general methods operating at the thread level—usually implemented in the operating system at page granularity, and task-oriented methods operating at the task level in parallel programming languages—typically in userland.

Starting with userland methods, it is possible to statically place arrays and computations to bring NUMA awareness to OpenMP programs [27, 2, 34, 32] or applications using TBB [26]. Such approaches are well suited to regular data structures and involve target-specific optimizations by the programmer. This is viable in some application areas, but generally not consistent with the performance portability and dynamic concurrency of task-parallel models.

ForestGOMP [5, 6] is an OpenMP run-time with a resource-aware scheduler and a NUMA-aware allocator. It introduces three concepts: grouping of OpenMP threads into bubbles, scheduling of threads and bubbles using a hierarchy of runqueues, and migrating data dynamically upon load balancing. Although the affinity between computations and data remains implicit, ForestGOMP performs best when these affinities are stable over time, and when the locality of unstable affinities can be restored through scheduling without migration. LAWS [11] brings NUMA awareness to Cilk tasks. It specializes into divide-and-conquer algorithms, assuming a one-to-one mapping between task trees and the memory regions accessed during task execution. LAWS put together a NUMA-aware memory allocator and a task tree partitioning heuristic to steer a three-level work-stealing scheduler, exploiting implicit information on the structure of data accesses and computation. While both ForestGOMP and LAWS show that such information can be exploited to increase locality, their limitations suggest that NUMA awareness is more natural and effective to achieve with the explicit task-data association we consider.

Among dependent task models, one of the most popular is the StarSs project [30], whose OMPSs variant led to dependent tasks in OpenMP 4.0. This model does provide an explicit task-data association. Yet we preferred OpenStream as it facilitates the privatization of data blocks communicated across tasks, and its first-class streams expressing dependences across concurrently created tasks, accelerating the creation of complex graphs on large scale NUMA systems. Our analysis of NUMA-aware placement and scheduling and the proposed algorithms would most likely fit other models with similar properties such as CnC [8] or KAAPI [16].

In earlier work [14], we introduced the work-pushing technique for task-parallel application that we extended in this paper. Furthermore, we proposed dependence-aware allocation, a NUMA-aware memory allocation technique that examines inter-task data dependences and speculatively allocates memory for the input data of a task on the node whose cores are likely to execute its producers. However, the techniques presented in this paper achieve better locality and performance since in some cases the prediction might fail and work-stealing might lead to remote write accesses.

Alternatively, one could argue that the operating system should be in charge of providing a NUMA-oblivious abstraction for parallel programs. Operating systems typically follow a first-touch strategy by default, in which a page of physical memory is allocated on the node associated to the core that first writes to the page. A common optimization is to migrate a page dynamically to the node performing the next write access. This strategy, referred to as affinity- or migrate-on-next-touch can be transparent [17] or controlled through system calls [25]. Dashti et al. proposed Carrefour [13], a kernel-level solution whose primary goal is to avoid congestion on memory controllers and interconnects. Carrefour detects affinities between computations and pages dynamically using hardware performance counters. Based on these affinities it combines allocation of pages near the accessing node, interleaved allocation and page replication for read-only data. However, the solution is suited for long-

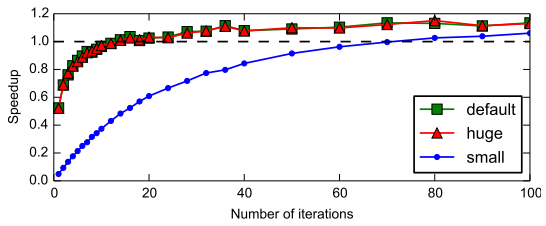


Figure 11: Speedup of page migration (synthetic benchmark)

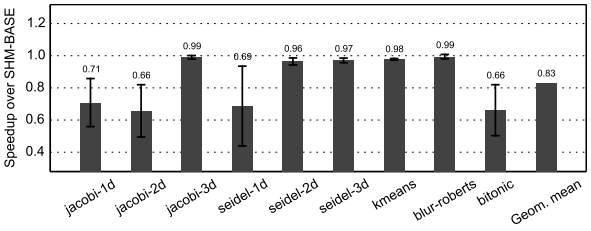


Figure 12: Speedup over static placement (SHM-BASE)

running processes as it uses sampling hardware counters [15] to determine data affinities, requiring high reuse of data for statistical confidence without high overhead [24]. Similar to Carrefour, AsymSched [24]—a combined user-space-kernel solution for data and thread placement—focuses on bandwidth rather than locality, taking into account the asymmetry of interconnects on recent NUMA systems. However, despite a highly efficient migration mechanism, the placement granularity is bound to a page and may not meet the requirements of task-parallel applications.

8. CONCLUSION

We showed that parallel languages with dependent tasks can achieve excellent, scalable performance on large scale NUMA machines without exposing programmers to the complexity of these systems. One key element of the solution is to implement communications through task-private data buffers. This allows for the preservation of a simple, uniform abstract view for both memory and computations, yet achieving high data locality. Inter-task data dependences provide precise information about affinities between tasks and data in the run-time, improving the accuracy of NUMA-aware scheduling. We proposed two complementary techniques to exploit this information and to manage task-private buffers: enhanced work-pushing and deferred allocation. Deferred allocation guarantees that all accesses to task output data are local, while enhanced work-pushing improves the locality of accesses to task input data. By combining hierarchical work-stealing with enhanced work-pushing, we ensure that no processor remains idle, unless no task is ready to execute. Deferred allocation provides an additional level of load-balancing, addressing contention on memory controllers. We showed that our techniques achieve up to 5× speedup over state of the art NUMA-aware solutions, in presence of dynamic task creation and changing dependence patterns. In a comparison with transparent static and dynamic allocation techniques by the operating system, we showed that our solution is up to 5.6× faster than interleaved allocation and that dynamic page migration is unable to cope with the fine-grained concurrency and communication of task-parallel applications.

Acknowledgments

Our work was supported by the grants EU FET-HPC ExaNoDe H2020-671578 and UK EPSRC EP/M004880/1. A. Pop is funded by a Royal Academy of Engineering University Research Fellowship.

9. REFERENCES

[1] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[2] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA machines. In *Supercomputing*. ACM/IEEE, Nov. 2000.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, Sept. 1999.

[5] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of openmp applications for multicore architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE.

[6] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, 38(5):418–439, 2010.

[7] F. Broquedis, T. Gautier, and V. Danjean. LIBKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Z. Budimlič, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Scientific Programming*, 18:203—217, 2010.

[9] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages,*

and Applications, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[11] Q. Chen, M. Guo, and H. Guan. LAWS: Locality-aware Work-stealing for Multi-socket Multi-core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM.

[12] J. Corbet. NUMA in a hurry, Nov. 2012.

[13] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 381–394, New York, NY, USA, 2013. ACM.

[14] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):30:1–30:25, Aug. 2014.

[15] P. J. Drongowski. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Advanced Micro Devices, November 2007.

[16] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 15–23. ACM, 2007.

[17] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*, pages 1–9, May 2009.

[18] Intel Corporation. Intel Math Kernel Library. `https://software.intel.com/en-us/intel-mkl`, accessed 01/2015.

[19] Intel Corporation. Threading Building Blocks. `https://www.threadingbuildingblocks.org/`, accessed 09/2015.

[20] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *Proceedings of the 2015 Usenix Annual Technical Conference*, USENIX ATC '15, pages 263–276, Berkeley, CA, USA, 2015. USENIX Association.

[21] A. Kleen. A NUMA API for Linux, Apr. 2005.

[22] M. Kong, A. Pop, L.-N. Pouchet, R. Govindarajan, A. Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Transactions on Architecture and Code Optimization*, 11(4):61:1–61:30, Jan. 2015.

[23] N. M. Lê, A. Pop, A. Cohen, and F. Z. Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, New York, NY, USA, February 2013. ACM.

[24] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 277–289, 2015.

[25] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA, 2005. ACM.

[26] Z. Majo and T. R. Gross. A library for portable and composable data locality optimizations for numa systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 227–238, New York, NY, USA, 2015. ACM.

[27] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta. Exploiting memory affinity in openmp through schedule reuse. *SIGARCH Computer Architecture News*, 29(5):49–55, Dec. 2001.

[28] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*, July 2013.

[29] `http://www.openstream.info`.

[30] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture*, 23(3):284–299, 2009.

[31] A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization*, 9(4):53:1–53:25, Jan. 2013.

[32] C. Pousa Ribeiro and J.-F. Méhaut. Minas: Memory Affinity Management Framework. Research Report RR-7051, 2009.

[33] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A programming model for deterministic task parallelism. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 7–12, New York, NY, USA, 2011.

[34] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD, pages 59–66, Washington, DC, USA, 2009. IEEE Computer Society.

[35] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users' Guide – QUeueing And Runtime for Kernels*, 2011. `http://ash2.icl.utk.edu/sites/ash2.icl.utk.edu/files/publications/2011/icl-utk-454-2011.pdf`, accessed 10/2014.