*Virtual Tasks for the PACLIB Kernel**

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

April 27, 1994

Abstract

We have extended the task management scheme for the parallel computer algebra package PACLIB. This extension supports "virtual tasks" (tasks that are not yet executable) which are created more efficiently than "real tasks" (tasks that are immediately scheduled for execution). Virtual tasks become real only when the system is idling or existing real tasks can be recycled. Consequently, the overhead for task creation and synchronization but also the memory requirements of a parallel program may be reduced. We analyze the system theoretically and experimentally and compare it with another virtual task package. In part our observations and conclusions are significantly different.

Keywords: Light-weight processes, lazy task creation, scheduling, shared memory computers, computer algebra.

*Supported by the Austrian Science Foundation (FWF) grant S5302-PHY "Parallel Symbolic Computation".

1 Introduction

The purpose of this paper is twofold: first it reports the extension of the task management scheme for a parallel programming package developed at our institute. Second it carefully investigates the semantic and performance consequences of this modification and compares them with the results reported for a system that was developed elsewhere with similar objectives in mind. In part our observations turn out to be significantly different.

The parallel programming package we refer to in this paper is the runtime kernel of the PACLIB system for parallel computer algebra [6]. This system represents the core of a long-term project of our group that deals with the systematic construction of a library of parallel algorithms for symbolic algebra and related areas. The PACLIB kernel efficiently provides light-weight concurrency on shared memory multi-processors with a programming interface that is considerably higher-level than comparable packages.

The PACLIB kernel runs under a conventional Unix operating system of which it only requires support for creating multiple processes that may access a global memory space. The kernel implements its own scheduler to distribute light-weight processes (called *tasks*[1]) among Unix processes (serving as *virtual processors*). Here we applied the facilities of the μSystem package [3] that we considerably modified and extended for our purposes.

The term "light-weight" implies that parallelism may be utilized at a rather low-level yielding a large number of tasks with small grain-size. While this is basically true, we (as other researchers before us) soon faced an obvious discrepancy. Parallel program development is considerably simplified by the ability to create a large number of light-weight tasks, but each task is subject to a more or less small performance penalty (the nature of this penalty will be discussed later in this paper). Since the number of physical processors is actually very limited (say 20), the large number of light-weight processes is not appropriately rewarded by reducing the computation time at the same scale. In practice, during the fine-tuning of a parallel program the number of tasks is therefore reduced at least to the same order of magnitude (say some 100) as the number of processors.

The techniques of *virtual threads* [10] or *lazy task creation* [13] were developed to overcome this discrepancy. Virtual threads are basically just descriptions of threads that may be created at a low cost but are themselves not yet executable. The idea is that most of these virtual threads will never become real at their own (thus avoiding the overhead) but their descriptions will be executed by other real threads that are already in existence. Consequently, the number of real threads will be reduced and their grain size will be increased.

E.g. a load based inlining technique may be used to reduce the creation of real tasks: When a "virtual fork" is executed, the current system load is investigated. Only if the load is currently low, the virtual fork is converted into a real fork. Otherwise the virtual fork is immediately executed as a function call. However, such a load balancing decision is somewhat hazardous: since it is only based on the current snapshot of the system load, processors might idle in the near future due to a lack of executable tasks. More sophisticated methods are required to effectively reduce the number of real tasks *without* sacrificing the benefits of concurrency.

Our work was inspired by and owes much to Küchlin's and Ward's virtual C thread package [10, 11] that was developed on top of the PARSAC-2 system [8, 9]. This system introduces the concept of S-threads (symbolic threads) which are based on C threads as implemented

[1] Other packages use the term *threads* to denote light-weight processes. In this paper we treat "tasks" and "threads" as synonyms but in context of the PACLIB system prefer the former.

in several operating systems [1, 18, 14] and standardized in a POSIX draft [7]. The result of this work was a carefully tuned and very efficient package for multi-threaded symbolic algebra that was used for implementing a variety of algorithms. We will repeatedly refer to this system and in large detail compare our results and conclusions with theirs.

Küchlin's and Ward's work was itself preceded by two other systems: WorkCrews [19] queues fork requests with the parent thread but makes it stealable by other processors. When the parent thread needs the result, it must either join the child thread (if it was stolen by another processor) or it executes the work request itself as a procedure call. Mohr, Kranz, and Halstead [13] introduce lazy task creation in the context of MultiLisp. The continuation corresponding to a forked thread may be stolen by another processor yielding the same overall effect.

2 The PACLIB Runtime System

To make this paper self-contained we give a short survey on the design of the PACLIB kernel. This design has been described in larger detail in [15, 16, 5].

The PACLIB kernel is a runtime system for light-weight concurrency on shared memory multi-processors. It has been designed for the parallel implementation of symbolic (in particular computer algebra) algorithms and integrates two free software packages:

- SACLIB [2] is a library of C functions that provides all fundamental objects and methods of computer algebra. The library is based on a runtime kernel for list processing with automatic garbage collection.
- The μSystem [3] is a library for light-weight concurrency on shared memory multiprocessors and workstations. The μSystem kernel distributes *tasks* (light-weight processes) among *virtual processors* which are UNIX processes that are scheduled by the native operating system.

We have applied and considerably extended the mechanisms of the μSystem to develop a suitable parallel programming interface for SACLIB. Most SACLIB functions implement purely mathematical functions that are entirely defined by their argument/result behavior. The PACLIB programming model [6] reflects this view:

t = `pacStart(`f`,` a_i`)`

creates a new task that asynchronously executes $f(a_i)$. The task handle t is a first-order object that can be passed to other functions/tasks and stored in any SACLIB data structure.

v = `pacWait(&`t`,` ts`)`

returns the result v of one of the denoted tasks ts and returns the handle t of the delivering task. If all denoted tasks are still active, `pacWait` blocks until a task terminates.

`pacWait` is a *non-deterministic* construct whose result in general depends on the system situation at runtime. Only when applied to a single task, its result is uniquely determined. The additional level of flexibility provided by non-determinism may help to considerably reduce the number of task blockings.

`pacWait` is also *non-destructive* i.e. it does not destroy the task whose result is delivered. As a consequence arbitrarily many tasks may asynchronously wait for the same set ts without

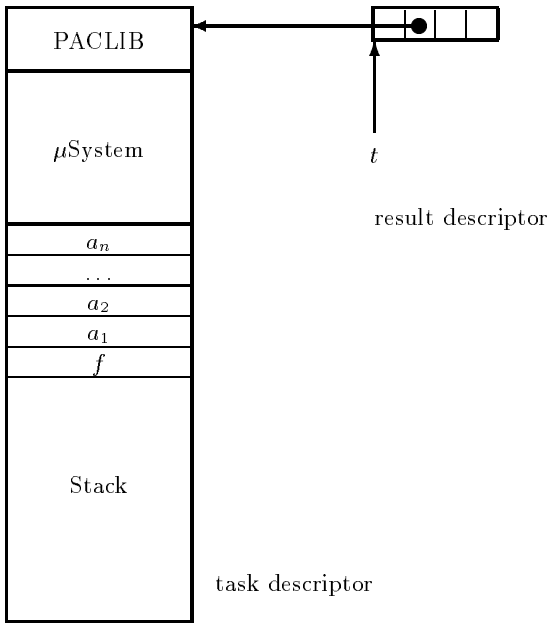

Figure 1: A PACLIB Task

interference. This feature distinguishes the PACLIB kernel from most other thread packages except for the MultiLisp futures [4].

To make this level of abstraction feasible, we have split the description of a task into two separate entities (see Figure 1):

1. The PACLIB *task descriptor* is an extension of the μSystem task descriptor. The μSystem task descriptor is approximately 100 bytes large, the PACLIB extension requires additional 40 bytes. However, most of the space of the descriptor is actually occupied by the task stack. The size of this stack is fixed but may be selected by the programmer before the creation of a task; currently a default value of 32 KB is used.
2. The PACLIB *result descriptor* is a short SACLIB list that contains a semaphore for mutual exclusion, a pointer to the corresponding task descriptor and either the result value itself or a queue of descriptors for those tasks that are blocked on an attempt to get the result.

`pacStart(`f`,`a_i`)` allocates a task descriptor and stores the function pointer f and the arguments a_i at the begin of the stack. A result descriptor is allocated and mutually linked with the task descriptor[2]; a reference to this result descriptor is returned as the task handle t. The new task executes a function `pacShell` that calls $f(a_i)$. When f has returned its result, `pacShell` stores the value in the result descriptor and awakes all tasks blocked on this descriptor. Finally `pacShell` erases the task link in the result descriptor, deallocates the task descriptor and terminates the task.

The large task descriptor therefore occupies space only as long as the task is executing. After termination only the small result descriptor remains active from where subsequent `pacWait`

[2]To simplify our figures, only the link from the result descriptor to the task descriptor (but not the inverse link) is depicted.

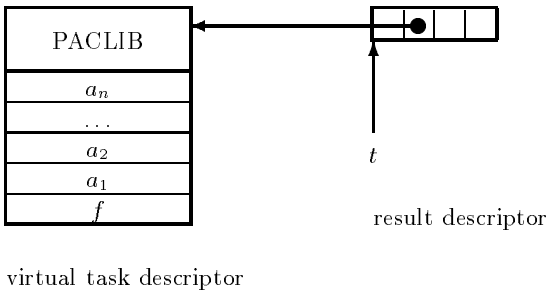

Figure 2: A Virtual PACLIB Task

calls will return the result without blocking. Since this descriptor is a SACLIB object, it is subject to the general garbage collection mechanism and together with the result value reclaimed when no task has the handle t any more. Hence, also task results do not occupy memory longer than necessary.

The PACLIB programming model also provides support for speculative parallelism, communication streams, and buffers [6]. The garbage collection mechanism is explained in large detail in [5]. The PACLIB kernel has been implemented on a Sequent Symmetry shared memory computer with 20 processors under the Dynix/ptx version of the UNIX operating system. It has been used for the implementation of a variety of computer algebra algorithms e.g. [12].

3 Virtual PACLIB Tasks

In the following subsection we describe the concept of virtual tasks as implemented in the PACLIB kernel. This implementation was inspired by the virtual S-thread package of Küchlin and Ward [10] but also contains a variety of original features. The main differences between both designs are discussed in Subsection 3.2.

3.1 Basic Description

Virtual PACLIB tasks are created by a function call

t = `pacVirtual`(f, a_i)

with the same interface and basic semantics as a `pacStart` call. The difference between both calls is the activation time of the task t: a call of `pacStart` immediately creates a full task descriptor and links it into the ready queue of tasks to be selected for execution.

`pacVirtual` however only allocates a virtual task descriptor as depicted in Figure 2. This descriptor consists of the PACLIB extension of the μSystem task descriptor and space for the function pointer f and arguments a_i. Virtual task descriptors have fixed size (about 80 bytes) and are obtained by locked access from a central pool (that is dynamically extended on demand). The descriptor is linked into a "virtual" scheduling queue that is distinct from the ready queue of "real" tasks.

The new result descriptor contains a link to the virtual task descriptor; the handle t returned by `pacVirtual` may participate in all operations like the handles of result descriptors that are returned by `pacStart` and thus connected to real tasks.

A virtual task is "realized" (i.e. transformed into a real task) on the following occasions:

1. **Idling:** The scheduler of a virtual processor finds the global queue of executable tasks empty. In this case it picks a descriptor from the virtual task queue and allocates a full-sized task descriptor with a stack. The contents of the virtual descriptor are transferred into the task descriptor and the link of the corresponding result descriptor is reset to this descriptor. The virtual descriptor is freed and the new task is activated.
2. **Termination:** A task is going to terminate after it has delivered its result. In this case the task tries to grip a descriptor from the virtual task queue. If such a virtual task is found, the task overwrites its own descriptor with the retrieved information, resets the link in the corresponding result descriptor, frees the virtual task descriptor and restarts its execution with the computation of the denoted function.
3. **Deterministic Wait:** A task calls `pacWait` for a single task that is still virtual. In this case, the task grips the virtual task, computes the denoted function itself, delivers its result into the corresponding result descriptor (and to all tasks waiting there) and continues execution.
4. **Non-deterministic Wait:** A task calls `pacWait` for several tasks some of which are still virtual. In this case, the task realizes all virtual tasks before it gets blocked. Otherwise, the semantics of `pacWait` would be effectively changed, since some of the virtual tasks might never become active (if other tasks do not terminate).

Case 1 is handled by the scheduler code executed in every virtual processor. Case 2 is managed by the `pacShell` function that every PACLIB task executes. Cases 3 and 4 are covered by the `pacWait` function. Consequently no change in the user code is necessary, i.e. the management details of virtual tasks are entirely hidden from the user. The extra runtime overhead in `pacShell` and `pacWait` consists of a simple test operation.

We will now discuss some of the consequences of our scheduling scheme for virtual tasks:

1. A real task is only created (unless explicitly by `pacStart`) when a processor would idle (or the semantics of the program would be changed) otherwise. The application of virtual tasks may therefore essentially reduce (a) the total number of real tasks ever created and (b) the maximum number of real tasks that exist at any moment.
2. Virtual tasks have lower priority than real tasks. While the available processor time is fairly scheduled among all real tasks on a preemptive basis, a virtual task becomes only active when some real task or some processor has no other choice (in particular if there is nothing else to do).
3. A task is only terminated when there is no more virtual task to be executed, hence the workspace of a task is reused as long as possible. This may essentially improve the data locality of a program by the reduction of its overall memory requirements (yielding a higher cache hit ratio and less disk paging).
4. In a deterministic wait, a task is only blocked when the task connected to the result descriptor is currently active. Otherwise, the corresponding task either has already delivered the result or is still virtual and gets "inlined" into the current task.

5. In a non-deterministic wait, none of the participating tasks is virtual. This would effectively change the behavior of the program since some of the virtual tasks might never (or at least too late) become active.

Since a task may be blocked only on a `pacWait`, the last two arguments show that virtual tasks do not change the termination semantics of a PACLIB program. In other words, no *virtual deadlock* is introduced by the scheduler.

3.2 Comparison to Virtual S-Threads

While there are several similarities, our concept differs in some important aspects from Küchlin and Ward's scheme. In their package virtual threads are transformed into real threads on two occasions:

1. When executing a `vthread_join`(t) operation (a deterministic destructive wait), the current thread checks whether t is still virtual. If this is the case, the current thread executes the virtual thread itself.
2. A scheduler thread wakes up from time to time to check the current load. If the number of active threads is below some threshold, the scheduler transforms an appropriate number of virtual threads into active threads. A copy of the scheduler code is also run by each thread as a side effect of a potentially load changing virtual thread call.

When a thread is going to terminate, this scheme does not reuse the thread for the execution of another virtual thread but its workspace is effectively deallocated. New real threads are later created when the scheduler decides that the current load is too low.

In our scheme tasks are never terminated if there still exist virtual threads to be executed. Furthermore we do not use a global load estimation scheme to start new real tasks. Instead every processor itself transforms virtual threads into real threads if it would idle otherwise.

A consequence of the different task scheduling policies is depicted in Figure 3 illustrating a divide-and-conquer algorithm where each task recursively creates two new tasks:

1. Without using virtual tasks at all, a real task is created for each node in the call tree; all non-leaf tasks are blocked waiting for the results of their children.
2. In Küchlin's and Ward's scheme, a single real task will typically execute a whole subtree (by inlining the corresponding virtual tasks); when this subtree has been processed, the task terminates.
3. In our scheme, a single real task may execute independent subtrees; when one subtree is processed (by inlining the corresponding virtual tasks), the terminating task will execute any other unprocessed (not necessarily sister) subtree.

In case of a divide and conquer algorithm, this difference does not matter under the assumption that there are $P = 2^p$ processors and the execution tree is perfectly symmetric and balanced with all tasks in the same level performing the same amount of work. Then in both schemes P real tasks will execute the P subtrees rooted at level p of the tree and terminate at the same time. Only in the unbalanced case our scheme may yield less tasks.

The distinction between both schemes becomes more compelling for linear recursive functions of the form

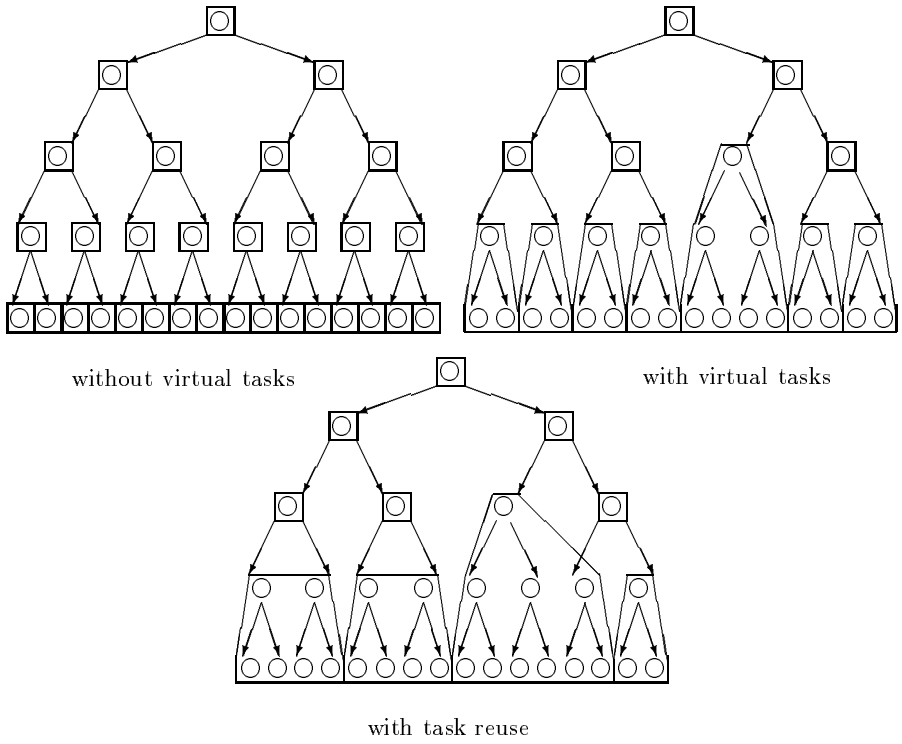

Figure 3: A Divide-and-Conquer Tree

```
f(x) = if (x == 0) then b else h(g(x),f(x-1))
```

We assume that on a call $f(n)$ the current task recursively starts n virtual tasks for the execution of the $g(x_i)$. In Küchlin and Ward's scheme, then $O(n)$ real tasks are created since each real task computing some $g(x_i)$ cannot inline any other virtual task. In our scheme, $O(P)$ real tasks are created (where P is number of processors) since each real task that has finished the computation of some $g(x_i)$ may continue with the computation of any other $g(x_j)$. A related problem with a master/slave parallelization is stated in [10].

4 Analysis of Features

What exactly are the main advantages of virtual tasks? Küchlin and Ward give a twofold motivation for their work [10]:

1. Virtual threads overcome the problem that the Mach operating system on their Encore Multimax only allows 1024 kernel threads at at time.
2. Virtual threads considerably reduce the overhead for thread creation and synchronization and thus allow tasks with smaller grain sizes.

While their original motivation mainly stems from the operating system limitation, Küchlin and Ward especially elaborate the second argument based on a recursive divide-and-conquer algorithm. In the following subsection we will investigate this argument in detail.

4.1 Minimum Grain Size of Tasks

In [10] a sorting algorithm is parallelized that follows the merge-sort paradigm but uses bubble-sort for short lists. The recursive creation of real threads has to be stopped at a certain cut-off point since the grain-size becomes to small. However all recursive calls can be spawned as virtual threads without significant loss of performance. Hence the authors consider the ability to avoid parallel cut-off points in recursive algorithms as the main advantage of virtual threads.

We have taken a closer look at the timings of the thread operations given in [10] and also at the timings of the corresponding PACLIB operations [17]. Since the underlying hardware is different, we roughly normalize all timings with respect to a function call with 4 arguments (= 1) which takes about 5 μs on our Sequent Symmetry and estimated 10 μs on Küchlin and Ward's Encore Multimax.

	Real S-thread	Virtual S-thread	PACLIB task	Virtual task
start	885 μs (89)	177 μs (18)	260 μs (52)	85 μs (17)
wait	585 μs (59)	257 μs (26)	340 μs (68)	85 μs (17)

This comparison is based on estimations and should be taken *cum grano salis*. It only shows that both packages essentially correspond in their performance. The application of virtual tasks considerably reduces the creation overhead and also the synchronization overhead. However we cannot help noting that in both packages this overhead is still an order of magnitude larger than the overhead for a simple function call.

In the case of a recursive divide-and-conquer algorithm where the grain-size of the leaf calls is essentially zero, this overhead will be still much too large to spawn virtual tasks for all recursive levels. Only when the leaf calls have a minimum grain size of more than say 200 function calls, the overhead becomes negligible.

In Küchlin and Ward's list sort example the leaf threads perform a bubble-sort on lists of length 10 which requires at least 500 machine instructions i.e. the equivalent of 100 function calls. This grain-size was much too small for S-threads but the reduced virtual thread overhead could apparently cope with it better. However we believe that smaller grain sizes are a problem for S-threads as well. Consequently virtual tasks may reduce the startup and synchronization overhead by a factor of 3–4 but we have to be aware that a significant granularity constraint still exists.

4.2 Maximum Number of Tasks: Real Tasks Only

A divide and conquer algorithm usually provides much more parallelism than physical processors are actually available. If we assume that in each level of the binary call tree one of the recursive calls is spawned as a real task while the other one is executed within the current task, a call tree of height n generates $N = 2^n$ real tasks in total.

We may now put the question how many of these tasks are in existence at a time, in particular what is the *maximum* number of tasks in existence. Let us assume that we have a parallel system with $P = 2^p$ processors where $p \ll n$. Let us further assume that scheduling is done non-preemptively using a FIFO queue for real tasks (no virtual tasks are used).

The main task recursively spins down the left-most path of the call tree starting n tasks for the right subtrees rooted in this path. These children behave like their parent by spinning

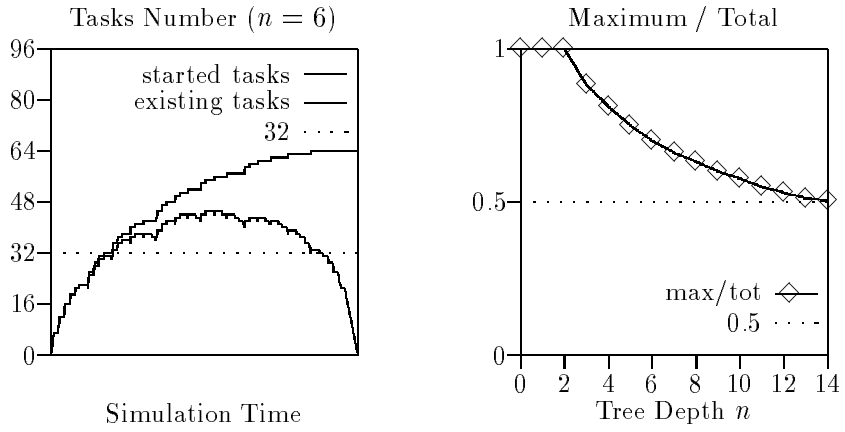

Figure 4: Simulation Results

down their left-most call paths and starting new tasks for their right subtrees. However, since we have only P processors, only the tasks started in the uppermost p tree levels find a processor for execution while the others only reside in the ready queue. Consequently only the oldest p of the n children of the main task are actually executing.

Now take a look at the behavior of one of the P executing tasks in charge of a subtree of height $m = n - p$. After the task has created m children, it solves its base case, steps one level up and gets blocked when it demands the result of its youngest child. However due to the FIFO scheduling the $m - 1$ elder sisters get scheduled before this child can execute and release its parent. The oldest child starts $m - 1$ tasks before it gets blocked, the second oldest child starts $m - 2$ tasks, and so on. Before the youngest child may execute, release its parent and terminate, $\sum_{i=1}^{O(m)} i = O(m^2)$ tasks have been created.

The parent task then combines its own subresult with the result of the terminated child, steps one level up and gets blocked again. Now all of the $O(m^2)$ existing tasks are scheduled creating $\sum_{i=1}^{O(m^2)} i = O(m^3)$ new tasks. After a few such iterations already a significant fraction of the total number of tasks has been created while only a few leaf tasks (and few tasks close to leaves) have terminated. The FIFO scheduling strategy effectively favors the breadth-first task expansion of the call tree over the termination of leaf tasks.

The left diagram in Figure 4 shows the results of a simulation of the FIFO strategy for a task executing a tree of height $n = 6$ with a single processor, which corresponds to the execution of one of the P task trees. One can clearly see on the left ascent the inverted m^2 curves resulting from the creation of $m, m - 1, \ldots, 1$ tasks separated by small cuts that denote the termination of a task. During most of the time more than 50% of the total tasks are in existence. The right picture shows that for increasing tree height the ratio between the number of existing tasks at a time to the total number of tasks converges to 0.5 i.e. the maximum task number is $O(2^n)$.

4.3 Maximum Number of Tasks: Virtual Tasks

What changes if we apply virtual tasks? Initially all virtual tasks are realized by idle processors. After saturation, each of the P real tasks inlines its virtual children in the order in which their results are required (youngest first) i.e. the call tree is processed depth first. Since this actually leads to a LIFO scheduling strategy for virtual tasks, the maximum number of virtual tasks is $O(n)$. On the other hand the maximum number of real tasks does *not* depend on the tree height n. If the call trees of all P tasks were of equal height, no more than P real tasks would be created at all.

In fact the realization of virtual tasks by idle processors takes considerably more time than the creation of virtual tasks. During any moment t before saturation, the number of real tasks N_t is not $2N_{t-1}$ (assuming unit time for the creation of a virtual task) but $N_{t-1}+N_{t-c}$ where c is the time required for the realization of a virtual task. Until all processors become saturated a virtual task has therefore some time advantage over its "nieces" (the children of its elder sisters) for getting realized. The call trees of the initial P real tasks are therefore rather unbalanced and the corresponding tasks complete execution at different times.

Every real task that has completely processed its call tree grips the first virtual task in the queue (which is the oldest child of some other real task) for further execution. The parent of the gripped task will be blocked later (because a task takes for processing its part of the call tree no longer than its oldest child). Since initially any blocked task causes some processor to idle, an additional virtual task is realized and more than P real tasks come into existence. Still the maximum number of real tasks is $O(P)$ since after a few iterations of task blockings and realizations every processor whose task becomes blocked finds already some other real task in the ready queue.

4.4 Why Reduce the Number of Tasks?

Küchlin and Ward's motivation to reduce the maximum number of tasks was to overcome a limitation of their operating system. In the PACLIB system such a limitation does not exist. Still we consider the reduction of the number of real tasks at a time as *the* advantage of virtual tasks which is of essential importance for the *runtime* of a PACLIB program and for its *applicability* in general.

An issue not elaborated by Küchlin and Ward is that on operating systems with virtual memory support the runtime of a program also heavily depends on its memory consumption and data locality. The PACLIB system equips a task by default with a 32 KB stack. The user may reduce this size but it will be difficult for a task to run with less than say 8 KB of space without risking stack overflow for recursive functions. Consequently 1000 real tasks already require 8 MB, 10000 tasks will eat up most of the available main memory. This large memory consumption together with regular context switches (by the preemptive scheduler) lead to a very low data locality in time and space.

First the cache hit ratio becomes rather low. Our Sequent Symmetry provides 512 KB cache memory per processor which suffices to keep the stacks of say 10 tasks in cache (we also have to consider the accesses to program text and heap memory). Context switches to additional tasks will lead to drastically increased cache misses and consequently heavy bus traffic.

For more than some hundred tasks, the memory consumption becomes so large that paging will set in: the operating system will start to transfer virtual memory pages that contain

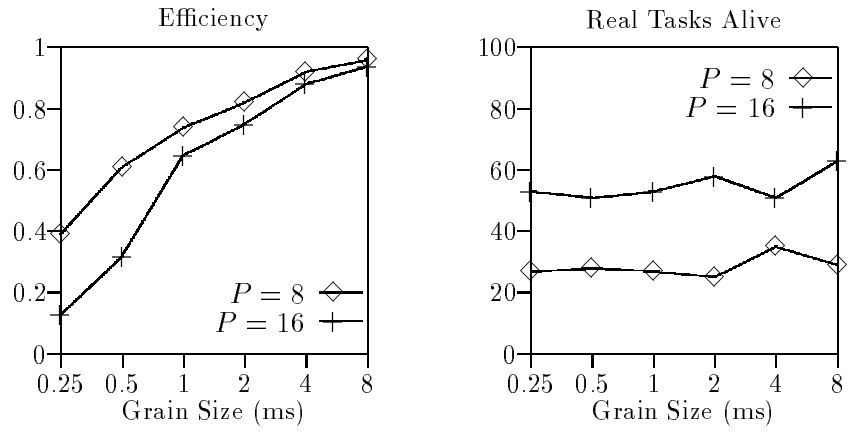

Figure 5: Minimum Granularity

stacks of recently not used tasks to the disk. Since each page fault requires say 10 ms (the equivalent of roughly 20000 function calls), the performance of the program will drastically degrade. It is illusory to start on a machine with 128 MB of main memory 10000 real tasks.

The situation changes if we apply virtual tasks; in the best case only as much real tasks exist at any time as physical processors. Even if somewhat more real tasks have to be created (due to other tasks being blocked), the virtual task mechanism in general allows us to start 10000 virtual tasks and still keep the memory consumption low. Furthermore, since the inlining of virtual tasks may lead to a FIFO scheduling strategy, the fraction of heap memory active at a time may reduce and make garbage collection more effective.

5 Experimental Results

We present the results of some benchmarks to justify the claims stated in the previous section. The timings were performed on a lightly loaded Sequent Symmetry multi-processor with 128 MB of physical memory and 20 processors i386 running at 16 Mhz.

5.1 Minimum Granularity

We use an artificial benchmark program to measure the minimum grain size that virtual PACLIB tasks permit. This program has a recursive divide-and-conquer structure where one recursive call is spawned as a virtual task and the task spins in the base case for a period of t ms. The program is called with $N = 12$ i.e. 4048 virtual tasks are created. Figure 5 displays the results of this benchmark executed with P processors:

1. The efficiency of the parallel program (compared to a corresponding sequential divide-and-conquer program) is acceptable for a grain size of $t \geq 1$ ms (which corresponds to 200 function calls). This is an order of magnitude less than the 10 ms granularity bound for real tasks [16] but still represents a significant constraint.

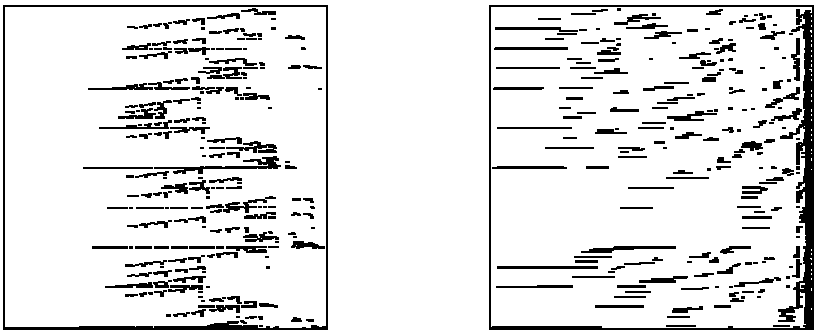

Figure 6: Profiling Data

2. During the program run at most 30 respectively 60 real tasks (about four times the number of processors) were in existence at a time independently of the grain size t. Consequently the total memory consumption of the program was limited to about 1.5 MB (with a 32 KB stack per task).

We also benchmarked a corresponding program with real tasks and a reduced stack size of 4 KB. This limited the maximum memory requirements of the program to 16 MB but still caused some page faults. Consequently the parallel program was in all cases 50–100 times *slower* than the sequential program.

Figure 6 visualizes profiling data generated for both program variants ($n = 10$, $t = 1$ ms) where the horizontal axis denotes the runtime. In the left diagram the lines represent virtual tasks where the long ones (actually sequences of short segments) exhibit real tasks that execute many virtual tasks. In the right diagram, lines represent real tasks; the long ones show where tasks were slowed down by page faults (note the different time scale). This drastically shows how the memory consumption of a program influences the runtime.

5.2 Parallel Bubble Merge Sort

We repeat the benchmark presented in [10] by parallelizing the SACLIB routine `LBIBMS` for sorting lists of single word integers. The algorithm is based on the divide-and-conquer merge-sort paradigm but lists of length 10 or less are sorted by a simple bubble-sort. Sorting a list of length N therefore requires approximately $L = \lceil \log_2 N - \log_2 10 \rceil$ recursion levels. Due to the long sequential steps for partitioning the input lists into two sublists and merging the sorted sublists into the result list, the achievable speedup is very limited. The results of several parallelization variants with $P = 8$ processors are presented in Figure 7:

1. `real`: a naive parallelization where in each recursion level one function call was replaced by `pacStart`. The benchmark was executed in a loop; the first iteration caused paging and was in all cases much slower than the sequential program. During the following iterations the virtual memory pages were in physical memory and the pure computation time could be measured. The maximum efficiency then is about 0.29.
2. `real(300)`: a parallelization with real tasks where a parallel cut-off point is introduced: for a list of length 300 or less no more tasks are created but the list is sorted sequentially.

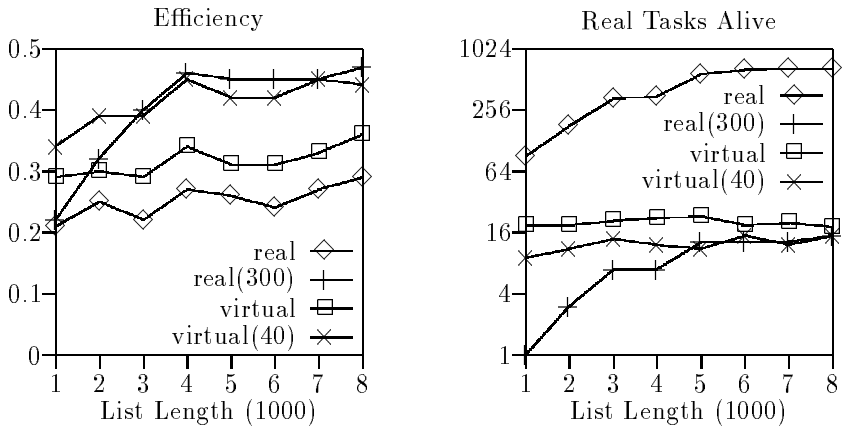

Figure 7: List Sorting

The maximum efficiency in this variant is about 0.47.

3. `virtual`: a naive parallelization like in `real` but with virtual tasks. The efficiency of this variant is slightly less than half way between `real` and `real(300)` giving a maximum of 0.36.

4. `virtual(40)`: like `virtual` but with a parallel cut-off point of 40 i.e. the two lowest levels of the call tree are executed sequentially. The efficiency is roughly the same as for `real(300)` but better for lists shorter than 3000.

A look at the maximum number of real tasks helps to explain above efficiency results:

1. The `real` program creates approximately 2^L tasks i.e. 128 tasks for $N = 1000$ and 1024 tasks for $N = 8000$. Our diagram shows that more than 50% (90–660) of these tasks were in existence at a time during the program run requiring about 20 MB of physical memory. The first iteration of the benchmark was heavily slowed down by paging; only the following ones were able to achieve some speedup.

2. The number of tasks created by `real(300)` is a factor of 30 smaller than in the previous program. Less than 8 tasks existed at a time for lists up to length 3000; here the parallel cut-off point hindered the utilization of available processors. For longer lists the maximum number of tasks was low but sufficient to saturate all processors; here this variant achieved the best efficiency.

3. The `virtual` variant kept the number of real tasks in existence at a level of 20. Unlike the `real` program, the memory requirements were therefore low and paging was avoided. More real tasks were created and in existence at any time as in `real(300)`; therefore processors were saturated with work better for shorter lists but efficiency was due to the small grain size lower for longer lists.

4. In the `virtual(40)` program, the grain size of virtual tasks was increased by a factor of about 5. In case of `virtual` the minimum task granularity was about 0.4 ms which is below the bound derived in the previous subsection. This granularity was now increased to roughly 2 ms giving optimum efficiency for shorter lists as well as for longer ones.

With increasing list length the maximum number of real tasks becomes essentially the same as in `virtual`.

These results show that in PACLIB the naive parallelization `real` was inefficient less because of the small grain size but much more because of its large memory consumption. Without paging the application of virtual PACLIB tasks increased the program efficiency by just 50%. However in practice a parallelization with a large number of real tasks causes many page faults and is therefore much slower.

We were not able to confirm Küchlin's and Ward's observation that the naive parallelization `virtual` was as efficient as the `real(300)` case. Instead we had to enlarge the granularity of virtual tasks by a small factor to achieve the same overall efficiency. Another less surprising observation is that virtual tasks with lower granularity than real tasks exhibit in smaller problems more parallelism and thus may achieve higher efficiency.

6 Conclusion

The application of virtual tasks reduces task creation and synchronization overhead by a factor of 3–4 and thus permits smaller grain sizes than real tasks. It is true that a main problem in parallelizing divide-and-conquer algorithms is to find the minimum grain size below which no more tasks should be spawned. However, the application of virtual tasks does not really solve this problem but only makes it somewhat less stringent.

Another parallel programming problem which is no less important is to spawn only that many tasks that can be easily kept in physical memory. The more tasks are started the more page faults occur and the slower the program becomes. Virtual tasks solve this problem effectively by reducing the memory consumption of tasks by a factor of almost 500. The programmer may therefore concentrate on the minimum granularity of tasks and spawn as many of them as appropriate for the algorithm. Consequently the main advantage of virtual tasks is the possibility to abstract from physical memory limits.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for Unix Development. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, August 1986.

[2] B. Buchberger, G. Collins, M. Encarnation, H. Hong, J. Johnson, W, Krandick, R. Loos, A. Mandache, A. Neubacher, and H. Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[3] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–964, September 1990.

[4] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[5] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of the SACLIB/ PACLIB Kernels. In A. Miola, editor, *DISCO 93 International Symposium on Design*

and Implementation of Symbolic Computation System, volume 722 of *Lecture Notes in Computer Science*, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[6] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.

[7] IEEE. Threads Extension for Portable Operating Systems (Draft 6). Technical Report P1003.4a/D6, IEEE, February 1992.

[8] Wolfgang Küchlin. PARSAC-2: A Parallel SAC-2 Based on Threads. In S. Sakata, editor, *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, Japan, August 1990. Springer, Berlin.

[9] Wolfgang Küchlin. The S-Threads Environment for Parallel Symbolic Computation. In R. E. Zippel, editor, *Second International Workshop on Computer Algebra and Parallelism*, volume 584 of *Lecture Notes in Computer Science*, pages 1 – 18, Ithaca, USA, May 1990. Springer, Berlin.

[10] Wolfgang Küchlin and Jeffrey Ward. Experiments with Virtual C Threads. In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, TX, December, 1992. IEEE Press.

[11] Wolfgang W. Küchlin and Jeffrey A. Ward. Experiments with Virtual Threads. Technical report, CIS Department, Ohio State University, Columbus, OH, September 1992.

[12] Carla Limongelli and Hans Wolfgang Loidl. Rational Number Arithmetic by Parallel P-adic Algorithms. In Jens Volkert, editor, *Parallel Computation — Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 72 – 86, Gmunden, Austria, October 4 – 6, 1993. Springer, Berlin.

[13] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[14] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. Technical report, Department of Computer Science, Florida State University, 1993. Also in: 1993 Winter USENIX — January 25 – 29, 1993, San Diego, California.

[15] Wolfgang Schreiner and Hoon Hong. A New Library for Parallel Algebraic Computation. In Richard F. Sincovec et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24, 1993.

[16] Wolfgang Schreiner and Hoon Hong. PACLIB — A System for Parallel Algebraic Computation on Shared Memory Computers. In Hussein M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, pages 56–61, Newport Beach, CA, April 14, 1993. IPPS '93.

[17] Wolfgang Schreiner and Hoon Hong. The Design of the PACLIB Kernel for Parallel Algebraic Computation. In Jens Volkert, editor, *Parallel Computation — Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 204–218, Gmunden, Austria, October 4–6, 1993. Springer, Berlin.

[18] Sun Microsystems. *System Services Overview, Lightweight Processes*, May 1988.

[19] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.