Virtual Tasks for the PACLIB Kernel*

Wolfgang Schreiner
schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

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.

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 [3]. 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* or *threads*) among Unix processes (serving as *virtual processors*). Here we applied the facilities of the μSystem package [2] that we considerably modified and extended for our purposes.

The term "light-weight" implies that a large number of tasks with small grain-sizes may be used. However each task is subject to a more or less small performance penalty (the nature of this penalty will be discussed later in this

* Supported by the Austrian Science Foundation (FWF) grant S5302-PHY.

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* [6] or *lazy task creation* [7] 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.

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 this load is 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 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 [6] that was developed on top of the PARSAC-2 system [5]. This system introduces the concept of S-threads (symbolic threads) which are based on C threads as implemented in several operating systems and standardized in a POSIX draft [4]. 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 preceded by two other systems: WorkCrews [9] 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 [7] 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

The PACLIB kernel [8] is a runtime system for light-weight concurrency on shared memory multi-processors. It has been designed for the parallel implementation of computer algebra algorithms and integrates two free software packages:

- SACLIB [1] 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 [2] is a library for light-weight concurrency on shared memory multi-processors. *Tasks* (light-weight processes) are scheduled among *virtual processors* implemented by UNIX processes.

We have applied and considerably extended the mechanisms of the μSystem to develop a suitable parallel programming interface for SACLIB. Most SACLIB functions are entirely defined by their argument/result behavior. The PACLIB programming model [3] 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.

`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 interference.

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

1. The PACLIB *task descriptor* contains besides other information the workspace (i.e. the stack) of the task. The size of this stack may be configured by the user; currently a default value of 32 KB is used.
2. The PACLIB *result descriptor* is a short SACLIB structure 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 a result descriptor and mutually links them; a reference to the 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` calls will return the result. 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.

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 [6] but also contains a variety of original features. The main differences 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 allocates a virtual task descriptor that only contains 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) only on

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 grab 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 grabs 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 executed in every virtual processor. Case 2 is managed by `pacShell` that every PACLIB task executes. Cases 3 and 4 are covered by `pacWait`. Consequently no change in the user code is necessary, i.e. the management details of virtual tasks are entirely hidden from the user.

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

1. A real task is only created when a processor would idle (or the program semantics might 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 `pacWait`, the last two arguments show that virtual tasks do not change the termination semantics of a program.

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 the 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.

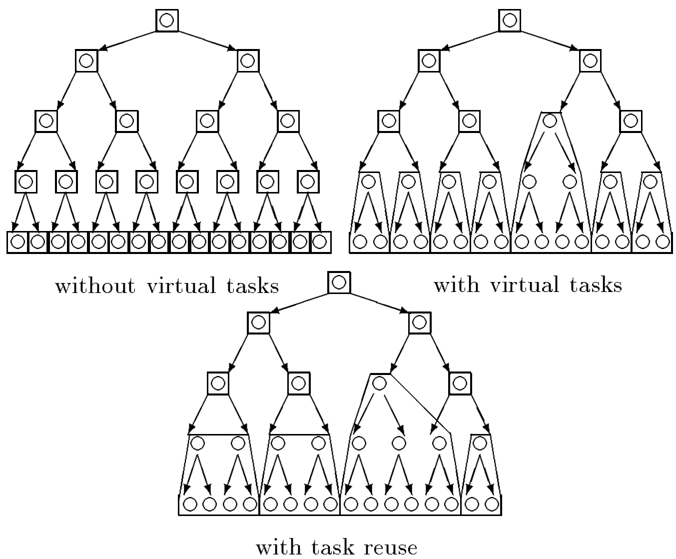

Fig. 1. A Divide-and-Conquer Tree

A consequence of the different task scheduling policies is depicted in Figure 1 illustrating a divide-and-conquer algorithm where each task recursively creates two new tasks:

1. Without using virtual tasks, 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 subtree.

The distinction between both schemes becomes more compelling for linearly recursive functions of the type

```
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 [6].

4 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.

4.1 Task Creation and Synchronization Overhead

In [6] 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 for thread operations given in [6] and at the timings of the corresponding PACLIB operations [8]. Since the underlying hardware is different, we compare the results to a function call with 4 arguments which takes about 5 μs on our Sequent Symmetry and estimated 10 μs on Küchlin and Ward's Encore Multimax. In Figure 2 these normalized values are given in parentheses.

	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)

Fig. 2. Timings of Task Operations

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 this overhead is still an order of magnitude larger than the overhead for a simple function call.

For 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.

4.2 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 3 displays the results of this benchmark executed with P processors:

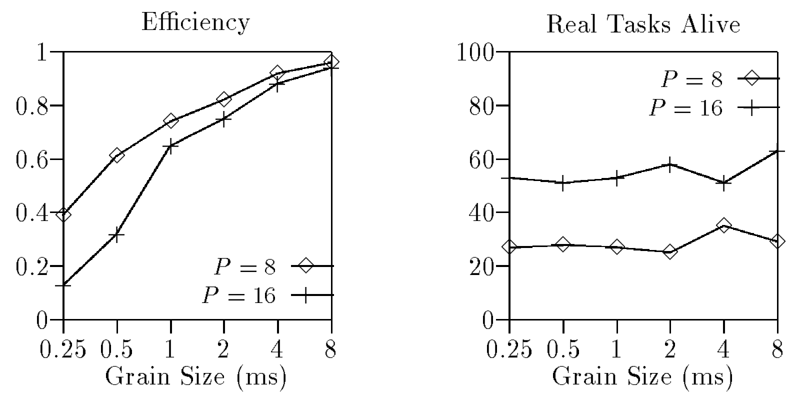

Fig. 3. Minimum Granularity of Tasks

1. The efficiency of the parallel program (i.e. the ratio S/P where S is the speedup over a 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 [8] but still represents a significant constraint.

2. During the program run at most 30 respectively 60 real tasks 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 note that with the application of virtual tasks the number of real tasks is about 3 times the number of processors. The reason that there are more virtual tasks than processors is that the initial realization of virtual tasks by idle processors takes more time than the creation of virtual tasks by the main task. Hence the children of the main task have some advantage over their "nieces" (i.e. the children of their elder sisters) and more than $\log_2 P$ children of the main task are initially executed. Some real tasks thus run shorter than others and grab on their termination virtual tasks in other subtrees for execution. This causes task blockings and the realization of additional tasks to keep processors busy.

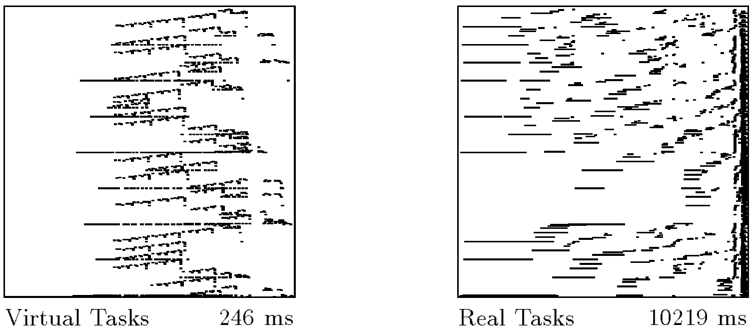

Fig. 4. Profiling Data

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. These results suggest that on the Sequent Symmetry the first access to a shared memory page always causes a page fault even if there is enough physical memory available.

Figure 4 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.

4.3 Parallel Bubble Merge Sort

We repeat the benchmark presented in [6] 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 are presented in Figure 5:

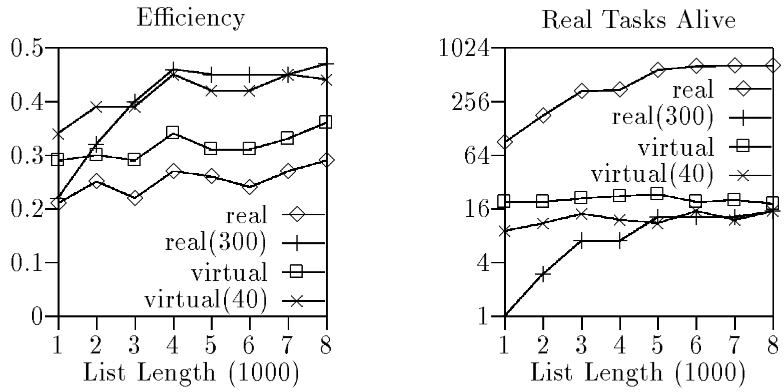

Fig. 5. List Sorting

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 page faults 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. The maximum efficiency in this variant is 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.

5 Conclusion

Virtual tasks reduce creation and synchronization overhead by a factor of 3–4 and thus permit smaller granularities than real tasks. It is true that a main problem in parallelizing (especially 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 makes it just less stringent.

Another problem which is no less important is to spawn only that many tasks that can be easily stored in physical memory. Memory is a valuable resource for which there is heavy competition between tasks. Moreover, the more space is used up by tasks, the less is available for the actual application data. Consequently, the more tasks are started, the more page faults occur and the slower the program becomes.

Virtual tasks solve these problems 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. Bruno Buchberger, George Collins et al. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.
2. 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.
3. Hoon Hong, Wolfgang Schreiner, et al. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.
4. IEEE. Threads Extension for Portable Operating Systems (Draft 6). Technical Report P1003.4a/D6, IEEE, February 1992.
5. 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.
6. 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.
7. 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.
8. 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.
9. M. T. Vandevoorde and E. S. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

This article was processed using the LaTeX macro package with LLNCS style