Manager-Worker Parallelism versus Dataflow in Distributed Maple*

Wolfgang Schreiner

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

December 13, 2000

Abstract

We analyze two implementation variants of a parallel computer algebra algorithm in Distributed Maple. The original (previously published) solution uses a manager-worker mechanism to control task scheduling, which requires an elaborate administration scheme. The new algorithm is based a dataflow approach where all tasks are immediately started, automatically scheduled by the runtime system, and implicitly synchronized by task dependencies; non-determinism is effectively applied to provide more potential for parallelism. It turns out that the new version is not only more declarative (closer to the mathematical problem description) but also more efficient than the original solution.

Keywords: parallel computing, computer algebra, algebraic geometry, declarative programming, task scheduling.

*Supported by the grants SFB F013/F1304 of the Austrian Science Foundation (FWF).

1 Introduction

We have developed Distributed Maple [4] as a portable and easy to use environment for implementing parallel computer algebra algorithms; so far, it has been successfully applied to the parallelization of various basic methods and complex applications in algebraic geometry [5, 9, 8, 2, 10]. Recently, we have analyzed the performance of the system on varous architectures in large detail [6].

A good deal of this effort has been to design the parallel algorithm, respectively to extract parallelism from a given sequential algorithm. However, even after the developer has devised the abstract algorithm, he is left with a number design decisions when constructing the concrete implementation: how to agglomerate independent activities to concurrent tasks, how to model the interactions between tasks, and how to schedule tasks for execution. This gives a wide spectrum of implementation strategies: from a program that explicitly controls task execution by a manager-worker scheme to a program that just creates all tasks and leaves the scheduling decisions to the runtime system. While Distributed Maple provides a rather declarative parallel programming model that favors the later approach, it is sufficiently general to support the former approach as well.

As a matter of fact, in our previous work the decision which style to pursue was essentially left to the taste of the developer: some algorithms were parallelized in a high-level declarative style, while others were implemented by low-level imperative mechanisms. The later strategy was typically chosen because of the opinion that keeping control over tasks yields a more efficient solution. However, due to the lack of comparative studies, there has up to now not been any well-founded basis for such a judgment.

In this paper, we are going to make good this omission by developing for a problem that has been previously solved in the manager-worker style of parallel computing a new implementation that is based on the dataflow principle: the new program starts all tasks as early as possible and leaves all scheduling and synchronization issues to the runtime system. Our basic concern was to make the program as simple and declarative as possible, i.e., to keep it close to the mathematical problem description. For this case study, we have chosen the problem of computing the Dixon resultant; this is part of the `neighbGraph` function of the computer algebra package CASA [3] and was previously parallelized in a manager-worker based style [8].

We have performed this work in preparation of a broader activity where we will compare parallel algorithms implemented in Distributed Maple with corresponding versions written for an environment that uses the para-functional language Glasgow Parallel Haskell GPH [1] to schedule computations among Maple kernels. For this purpose, we have already developed a Haskell-Maple interface [7]; a corresponding GPH-Maple environment is under preparation.

The remainder of this paper is organized as follows: in Section 2, we state the problem and describe the sequential algorithm; in Section 3, we repeat the original solution presented in [8]; in Section 4, we introduce a revised solution which is in Section 5 compared to the original one; in Section 6 we present the conclusions we draw from this case study.

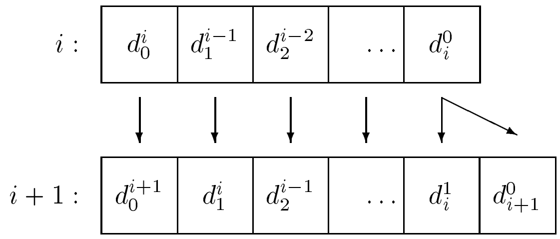


Figure 1: Computation of Derivatives of Total Order $i+1$

2 Problem and Sequential Algorithm

Let p be a bivariate polynomial with rational coefficients x and z, i.e. $p \in \mathbb{Q}(x,z)$, and let $d_u^v \in \mathbb{Q}(x,z)$ denote the partial derivative $\frac{\partial p^{u+v}}{\partial x^u z^v}$ of total order $u+v$. Our goal is to compute a sequence of univariate polynomials $[b_i]_{i=0}^n$ where $b_0(x)$ is the greatest square free divisor (gsfd) of $p(x,0)$, $b_{i+1}(x)$ is the greatest common divisor (gcd) of $b_i(x)$ and of all $d_v^u(x,0)$ with $u+v=i+1$, and n is the smallest number such that $\deg b_n = 0$.

In the sequential algorithm, the partial derivatives of total order $i+1$ are generated from (and overwrite) the derivatives of order i as illustrated in Figure 1:

$\boldsymbol{(b,n)}$ **:= derivatives**$\boldsymbol{(p)}$
  $i := 0$
  $d_0 := p$
  $b_0 := \text{gsfd}(d_0(x,0))$
  while $\deg(b_i) \neq 0$ **do**
    $d_{i+1} := \frac{\partial d_i}{\partial z}(x,z)$
    for j **from** 0 **to** i **do**
      $d_j := \frac{\partial d_j}{\partial x}(x,z)$
    $b_{i+1} := b_i$
    for j **from** 0 **to** $i+1$ **do**
      $b_{i+1} := \text{gcd}(b_{i+1}, d_j(x,0))$
    $i := i+1$
  $n := i$
end

3 Manager-Worker Parallelism

The basic parallelization idea is to organize the computation of all d_u^v in a triangular matrix as shown in Figure 2: each line i contains all d_u^v with $u+v=i$, each column j contains all d_j^v, i.e., the matrix holds at position (i,j) the derivative d_j^{i-j}. We may thus compute all those positions (i,j) with $i \geq j$ in parallel whose data dependencies have been resolved, i.e., for which the result

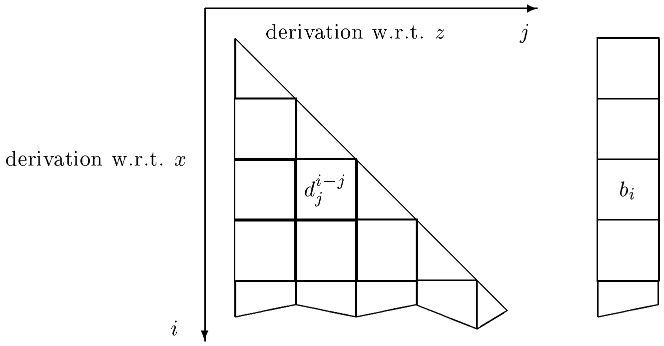


Figure 2: The Matrix of Partial Derivations

at position $(i-1, j)$ is available (if $i > j$) respectively the result at position $(i-1, j-1)$ is available (if $i = j$).

In order to allow an efficient implementation, the algorithm must increases the *granularity* of the parallel computation by letting each task compute multiple elements of the matrix: we partition the triangular matrix into square blocks of m^2 elements (for some blocking factor m) and let each task (i, j) compute the partial derivatives of the block with upper left corner (i, j). The blocks along the diagonal boundary of the triangular matrix are themselves triangular such that the corresponding tasks only need to compute $\frac{1}{2}(m^2 - m)$ elements.

All tasks are created by a manager program which itself computes iteratively the d_u^0, i.e., the derivatives along the diagonal boundary of the matrix. When the manager program has computed all those derivatives that represent the diagonal boundary of a triangular block, it starts a corresponding task that computes the values of this block. Whenever a task has terminated, the manager program starts a new task for computing that square block that is adjacent to the lower boundary of the result block.

Actually, the result of a task need not be the values of all d_u^v in the corresponding block because we are only interested in

1. the last line of the block which is required by the task computing the adjacent block (this result need not be returned to the main program but can be put into a shared space from which the other task can retrieve it);
2. the greatest common divisor of each line of the block which is required to compute the greatest common divisor of the whole matrix line (this result is returned to the manager program).

Since the gcd is commutative and associative, the program may receive in any order the results computed by the tasks of line i and combine them with the current value of b_i. In a final step, b_{i+1} is then combined with b_i.

We utilize the p processors of our machine by the following *scheduling strategy*: Initially, the manager program creates tasks for computation of the first p triangular blocks. Whenever a task terminates, we "enable" the adjacent square block whose computation thus becomes possible; if the terminated task has computed a triangular block (and it was not one of the p initial tasks), we also enable the subsequent triangular block. Among all enabled blocks, we choose a block with minimum line index (its computation may make the computation of blocks with larger indices superfluous). When the termination criterion is detected in line i, only those tasks will be started that operate on lines with indices less than i; when no more task is active, the algorithm terminates.

The algorithm can be formalized as follows:

(b, n) := derivatives(p)
  $T := \{\text{task } (im, im) : i = 0 \ldots p-1\}$
  $n := \deg(p)$
  while $T \neq \emptyset$ **do**
    wait for some task $(i, j) \in T$ and remove it from T
    update $b_i \ldots b_{i+m-1}$ and n
    enable $(i+m, j)$
    if $i = j$ **then** enable $(i+m, j+m)$ **end**
    disable all (i, j) with $i \geq n$
    if there is some enabled (i, j) with minimum i **then**
      disable it and add task (i, j) to T
    end
  end
  for i **from** 0 **to** $n-1$ **do**
    $b_{i+1} := \gcd(b_i, b_{i+1})$
  end
end

We have implemented this algorithm in Distributed Maple as part of a parallel version of CASA's `neighbGraph` algorithm. Figure 3 illustrates by three examples the dynamic behavior of the implementation (with block size $m = 2$) on 20 Linux PCs with various processors connected by switched 100Mbit Ethernet lines: the left "machine diagram" displays on the vertical axis each machine participating in the session; each line in the diagram represents a task executed on a machine at a particular time. The right "utilization diagram" lists on the vertical axis the number of machines busy at a particular time. The sequential execution of these examples took on a PIII@450MHz PC 552s, 198s, and 1798s respectively; the corresponding parallel computations took 27s, 14s, and 61s.

While the parallelization achieved significant speedups (partially gained by the fact that the modified order of gcd computations turns out to be more efficient than the one in the sequential algorithm), the utilization diagrams show that there is much room for improvement: rarely all 20 machines are busy, in average only about 50% of the computing resources are actually utilized. Apparently, the manager program is not able to issue parallel tasks at a rate that is sufficiently high to provide idle machines with work, i.e., the explicit task scheduling becomes a performance bottleneck. As illustrated by Figure 4, it does not help to increase the block size to say $m = 4$: while the manager program is then no more the bottleneck, simply too few tasks are generated to saturate all processors and the overall execution time increases to 36s, 15s, and 69 s, respectively.

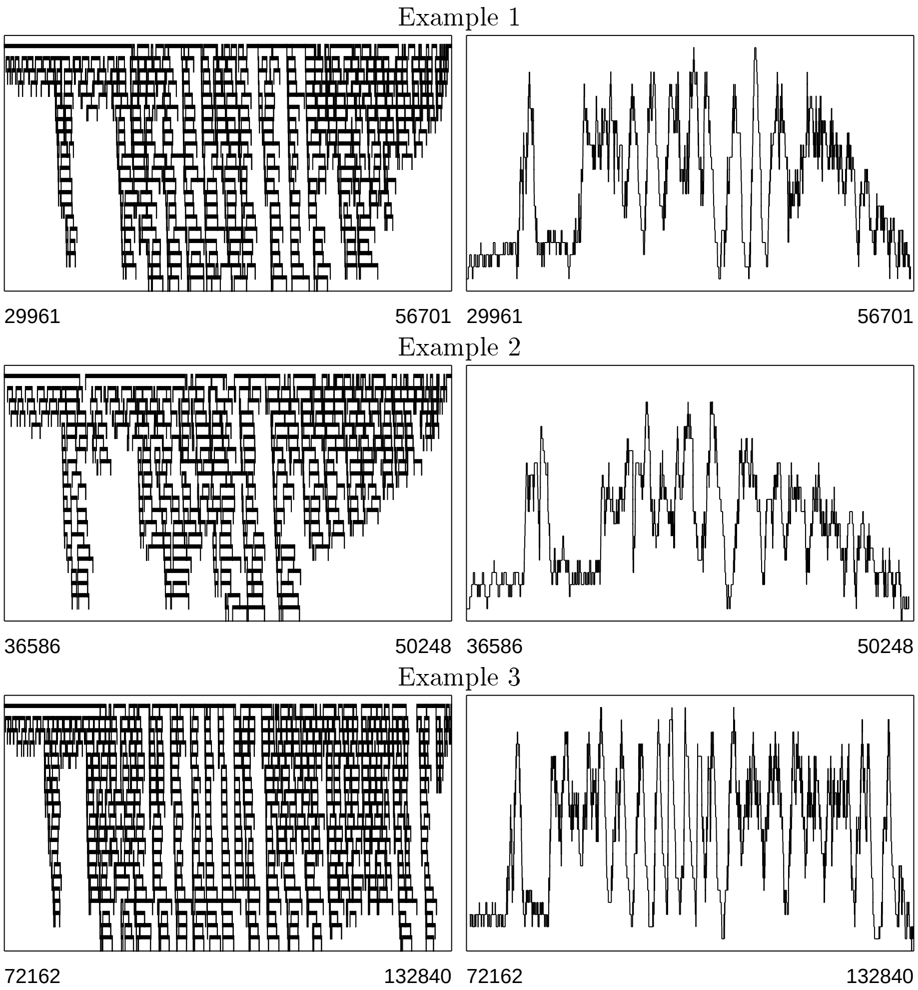


Figure 3: Manager-Worker Parallelism (Block Size $m = 2$)

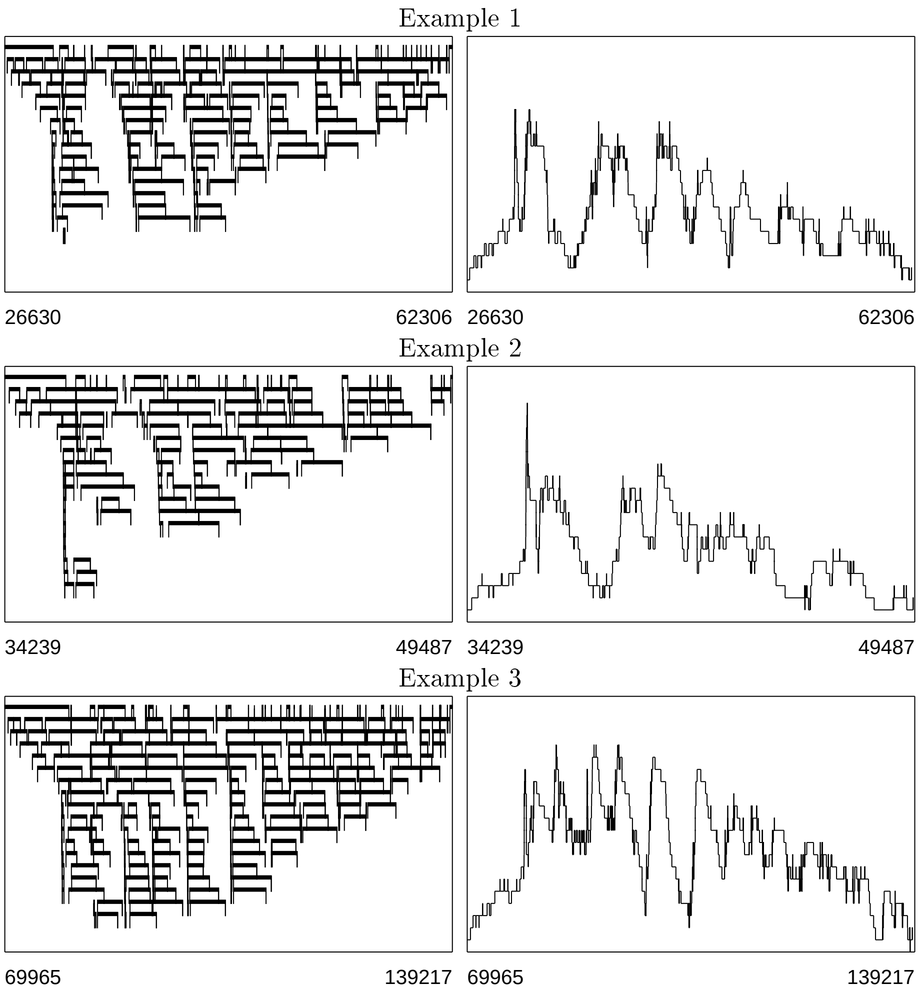


Figure 4: Manager-Worker Parallelism (Block Size $m = 4$)

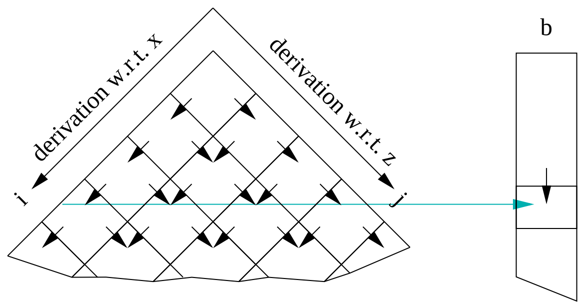


Figure 5: Data Dependencies

4 Dataflow Parallelism

In this section, we are going to explore an alternative way of exploiting the parallelism of the problem stated in Section 2. In contrast to the original solution, we are now going to investigate an approach based on the *dataflow* principle of computation where tasks are implicitly scheduled for execution whenever their data dependencies have been resolved. Having recognized the utilization problem, we will also try to exhibit all parallelism inherent in the problem. In a nutshell, we are now heading for a *declarative* approach where the manager program only describes the collection of tasks that have to be executed and leaves the imperative details to the runtime system. Our goal is to have a program of bigger *elegance* that shall also yield good *performance*.

We start the new design by reorganizing the decomposition of the triangle matrix of partial derivatives as shown in Figure 4: we partition the matrix into uniform square blocks of size m^2 (for some blocking factor m) such that each block can be identified by the coordinate (i, j) of its upper point denoting d_i^j and contains all d_u^v with $0 \leq u < i + m$ and $0 \leq v < j + m$. Once d_i^j is known, all other elements of the block can be determined by derivation with respect to x or z, respectively. Like in the original algorithm, we can compute within a block the gcd of all elements d_u^v with same sum $u + v$ which then contributes to the computation of the result vector b_{u+v}. Unlike in the original solution, the lines and columns of each block run diagonal to the computation of the derivatives as sketched by the grey arrow in Figure 5.

Since all elements in a block can be computed from the first element d_i^j, we can start the computation of a block when one of the following condition holds :

- $i = 0$ and d_i^{j-1} is known.
- $j = 0$ and d_{i-1}^j is known.
- $i > 0$ and $j > 0$ and
  - d_i^{j-1} is known *or*
  - d_{i-1}^j is known.

In other words, while a block $(0, j)$ at the right boundary of the triangle matrix depends on its unique upper neighbor $(0, j-m)$ (because it requires d_0^{j-1}) and a block $(i, 0)$ at the left boundary depends on its unique upper neighbor $(i-m, j)$ (because it requires d_{i-1}^j), a block (i, j) in the inner of the matrix depends on just *one* of its two upper neighbors $(i, j-m)$ and $(i-m, j)$ and can be computed whenever one of these neighbors has delivered d_i^{j-1} or d_{i-1}^j.

Let n_x be the degree of p in x, let n_z be the degree of p in z, and define $n := \min(n_x, n_z)$. Since $d_{n_x}^j$ and $d_i^{n_y}$ are constant, we know that b_n (the gcd of b_{n-1} and of all d_u^v with $u+v = n$) is constant and that therefore the computation terminates with some $k \leq n$. We therefore need to create only tasks (i, j) for which $i + j \leq n$; in the course of the computation it may turn out that $k < n$ and that therefore those tasks (i, j) with $i + j > k$ are not required any more and may be stopped.

We formalize our algorithm using the auxiliary definition

$$\text{tasks}(k) := \{(i, j) : i + j = k \wedge i \bmod m = 0 \wedge j \bmod m = 0\}$$

for all task positions (i, j) such that $i + j = k$:

$\boldsymbol{(b, n)}$ **:= derivatives**$\boldsymbol{(p)}$
  $n := \min(n_x, n_z)$
  for k **from** 0 **to** n **by** m **do**
    for (i, j) **in** tasks(k) **do**
      if $i = 0 \wedge j = 0$ **then** $t_{i,j}$:= **start** task$(i,\ j,\ m,\ p)$
      else if $i = 0$ **then** $t_{i,j}$:= **start** $\text{task}_{\text{Right}}(i,\ j,\ m,\ t_{i,j-1})$
      else if $j = 0$ **then** $t_{i,j}$:= **start** $\text{task}_{\text{Left}}(i,\ j,\ m,\ t_{i-1,j})$
      else t_i := **start** $\text{task}_{\text{Inner}}(i,\ j,\ m,\ t_{i,j-1},\ t_{i-1,j})$
    end
  end
  for k **from** 0 **to** n **by** m **do**
    $tset := \{t_{i,j} : (i, j) \in \text{tasks}(k)\}$
    while $tset \neq \emptyset \wedge n = \min(n_x, n_y)$ **do**
      t := **select** $tset$
      $(\_, \_, g)$:= **wait** t
      update $b_{i+j} \ldots b_{(i+m)+(j+m)}$ and n
      tset := tset-$\{t\}$
    end
  end
  for k **from** $n + m$ **to** $\min(n_x, n_y)$ **by** m **do**
    for $(i, j) \in$ tasks(k) **do stop** $t_{i,j}$; **end**
  end
  for $t \in tset$ **do stop** t **end**
end

The manager program starts the tasks in some order compatible with the task dependencies and passes to each task (apart from the initial one which receives the input polynomial p) the references to those tasks that the new task potentially depends on. Then the program waits for all tasks (i, j) in the order of increasing $i + j$, combines the computed gcds with the vector b and signals by updating n whether the computation can be prematurely terminated. All tasks whose results are not required any more are then stopped.

The initial task computes, starting with p, the first square of derivatives and returns the "boundary derivatives" d^j_{i+m} and d^{j+m}_i (for use by the neighbor tasks) and the gcd of the derivatives (for the manager program):

$\boldsymbol{(p_1, p_2, g)}$ **:= task**$\boldsymbol{(i,\ j,\ m,\ p)}$
  compute all d^v_u with $i \leq u < i+m \wedge j \leq v < j+m$
  $(p_1, p_2) := (d^j_{i+m}, d^{j+m}_i)$
  for k **from** 0 **to** $2m$ **do**
    $g_k := \gcd\{d^v_u : i \leq u < i+m \wedge j \leq v < j+m \wedge u+v=k\}$
  end
end

The other tasks can then be formalized with the help of above description:

$\boldsymbol{(p_1, p_2, g)}$ **:= task**$_{\textbf{Left}}\boldsymbol{(i, j, m, t)}$
  $((p, \_), \_) :=$ **wait** t
  $(p_1, p_2, g) := \mathrm{task}(i, j, m, \frac{\partial p}{\partial x})$
end

$\boldsymbol{(p_1, p_2, g)}$ **:= task**$_{\textbf{Right}}\boldsymbol{(i,\ j,\ m,\ t)}$
  $((\_, p), \_) :=$ **wait** t
  $(p_1, p_2, g) := \mathrm{task}(i, j, m, \frac{\partial p}{\partial z})$
end

$\boldsymbol{(p_1, p_2, g)}$ **:= task**$_{\textbf{Inner}}\boldsymbol{(i, j, m, tr, tl)}$
  $t :=$ **select**$\{tr, tl\}$
  if $t = tr$ **then**
    $(p_1, p_2, g) := \mathrm{task}_{\mathrm{Right}}(i, j, m, t)$
  else
    $(p_1, p_2, g) := \mathrm{task}_{\mathrm{Left}}(i, j, m, t)$
  end
end

An "inner" task selects non-deterministically the result of one of the two neighbor tasks (whichever happens to be available first) and, depending on the selection, proceeds with the derivation in the style of a task on the right boundary respectively on the left boundary.

As an optimization, a task (i, j) actually needs not return the boundary derivatives d^j_{i+m} and d^{j+m}_i to the manager program (which is only interested in the gcd vector g); instead it may put the results into a shared space from which the other tasks may retrieve then (actually a task should do this as early as possible such that subsequent tasks become enabled even *before* the predecessor task has terminated). Consequently, each task needs not receive as arguments references to other tasks but only the locations of shared data where it may find the boundary derivatives when they have been computed. The main program may create these locations and pass to each task the location of its own result and the locations of the results it depends on. To avoid race conditions between the producer and the consumer of a shared data item, such an item needs to be *self-synchronized*: as long as it is empty, the consumer task gets automatically blocked on an attempt to retrieve its content; the consumer task is released as soon as a producer task provides the content.

The differences of the new algorithm to the original one can now be summarized as follows:

1. **Matrix Decomposition:** the new algorithm uses a more regular decomposition as the original one which dealt with two different block shapes ("triangular" and "square"). This has the advantage that each task computes the same number of derivatives m^2 (m^2 and $\frac{1}{2}(m^2-m)$ in the original solution) at the price that each task returns a gcd vector of length $2m-1$ (m in the original solution).

   However, the new decomposition strategy is not a characterizing feature; both strategies can be applied in both algorithms.

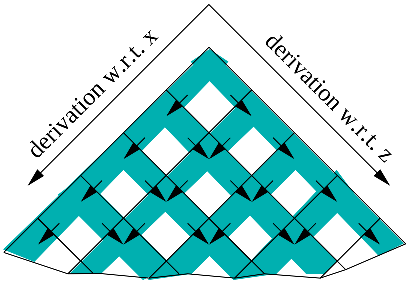
2. **Task Synchronization:** the new algorithm uses implicit synchronization based on task dependencies. The algorithm thus becomes much simpler than the original solution where dependencies had to be explicitly maintained in order to schedule exactly one task per processor at every time. Thus the new algorithm should in principle be able to utilize more parallelism and overcome the utilization problem discussed in the previous section. On the other side, creating many tasks at once (many of which may become activated and blocked) means also larger memory requirements, which may become a problem.

   In a nutshell, greater simplicity and (hopefully) better processor utilization at the price of larger memory requirements are the main distinguishing feature between both solutions.

3. **Result Values:** in the new algorithm, each task returns two derivatives and a gcd vector as a result; the old algorithm returned a vector of derivatives together with the gcd vector.

   This is an inefficiency in the original algorithm that was not previously recognized because of a focus on the behavior of the sequential algorithm where all but one derivatives in a line are constructed from the previous line by derivation with respect to variable x. It would have also sufficed to communicate a single derivative (the left lower corner of the triangle/matrix) from one task to its successor, since the fist line of each square block can be computed by derivation with respect to z.

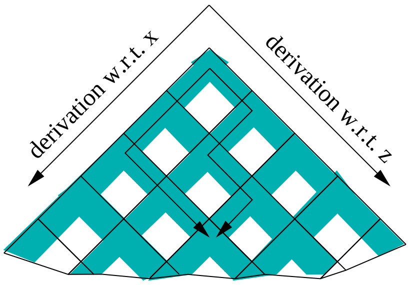
4. **Result Timing:** A task may put the computed boundary polynomials into a shared data space and thus enable the computation of two neighboring tasks tasks *before* it performs the remainder of its computation which is not required to enable other tasks. In this way, tasks may become active earlier and the utilization increase faster than in the original algorithm. As an illustration, the non-shaded areas in the following diagram denote those elements of the derivation matrix which are *not* required to enable all tasks to compute their results:



Again, a similar optimization might in principle have been performed in the original algorithm too.

5. **Non-Determinism:** the new algorithm does for inner tasks not specify which of the two possible predecessor tasks returns the boundary polynomial that enables the further execution of the task.

   An inner task may thus be enabled by different paths of task dependencies as illustrated by the following diagram that shows two possible paths that lead to the enabling of a particular task:



6. **Speculation:** the new algorithm exhibits more speculation than the original one, i.e., it starts more tasks whose result may turn out to be not required any more (and should be subsequently stopped). In the original algorithm, on each machine at most one task was active of a time, such that for p machines at most $p-1$ tasks may other $p-1$ tasks become superfluous. In the new algorithm, about $\frac{1}{2}(n/m)^2$ (i.e. *all*!) tasks are initially created before starting to check for termination.

   Thus, depending on the actual input, many of the created tasks may become superfluous. However, as shown in the next section, in practice most of the task results are actually required and only few tasks at the bottom of the triangle are computed in vain.

So far for the theoretical analysis of the algorithm. As discussed above, several optimizations of the new algorithm could have also been performed in the original algorithm but were not discovered at that time because of a too close focus on low level scheduling details. How far the speculated differences between original and new algorithm are actually relevant in practice, is experimentally investigated in the following section.

5 Experimental Results

We have implemented in Distributed Maple the algorithm described in the previous section (including the optimization of using shared data spaces for inter-task communication). The implementation follows closely the pseudo-code in the previous section. The new implementation required about 140 lines of Maple code (about 70 lines for the main program) while the original one required 200 lines (140 for the main program); i.e. the size of the source code was reduced by 30% (50% for the main program).

Implementation In the new implementation we cannot use the Distributed Maple call `dist[start]` to create the tasks and schedule them among the nodes of the distributed session. The reason is that, whenever a task is blocked by waiting for a result, the runtime system schedules another task for execution on the respective kernel. Multiple tasks on a kernel are executed last-in first out, i.e., the most recently scheduled task has to terminate before a previously suspended task can resume execution. However, in our algorithm, a task depends on another tasks passed to it as an arguments, i.e., a task executed on a Maple kernel may depend on another task that has *before* started execution on the same Maple kernel and was suspended.

For such situations, we have introduced the operation `dist[process]` which uses a free Maple kernel for the execution of the denoted task and consequently has to start on a Distributed Maple node additional Maple kernels to meet this requirement. The additional kernels are retained after termination of the executed task such that they can be later used for processing other `dist[process]` calls. As a consequence, more than one Maple kernel may be active on a Distributed Maple node at a time which increases the memory requirements on the corresponding machine. However, in the examples benchmarked below, only the frontend node (where the main program is executed) used more than kernel: up to five kernels were here active at the same time, i.e., on this node at most five tasks at a time were suspended and resumed.

Behavior and Utilization We have benchmarked the program in our local network with 20 Linux PCs with various Pentium class processors connected by switched 100MBit Ethernet lines. As discussed in [6], this class of architectures (Beowulf clusters") is currently the most suitable one for Distributed Maple applications. We have executed the sample benchmark problems as described in Section 3 and illustrate in Figures 6 and 7 the executions with block sizes $m = 4$ and $m = 5$ (those that gave best results).

All in all, the algorithm proceeds in the following phases:

1. all tasks are started and get immediately blocked (short activities on the left boundary of the machine diagram followed by empty space),
2. the first task is executed and lets other tasks resume execution which in turn let other tasks resume while the early tasks terminate again (task gap in the upper left part of the diagram);

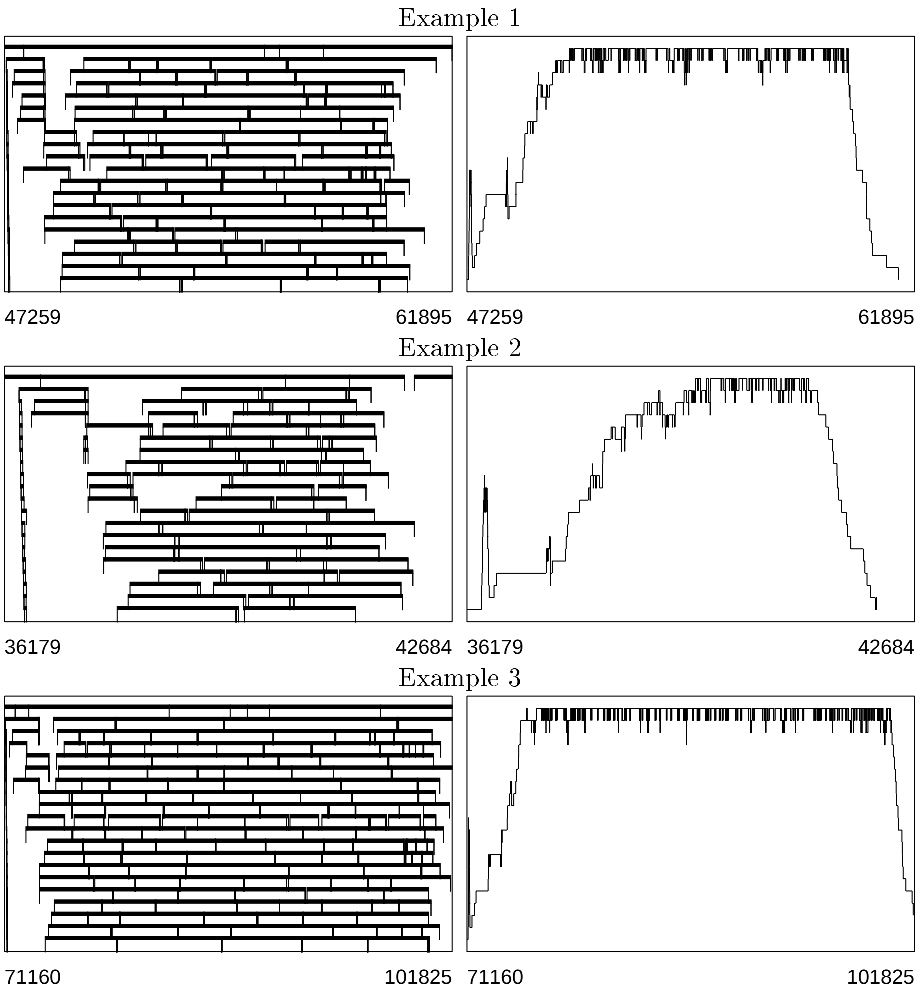


Figure 6: Dataflow Parallelism (Block Size 4)

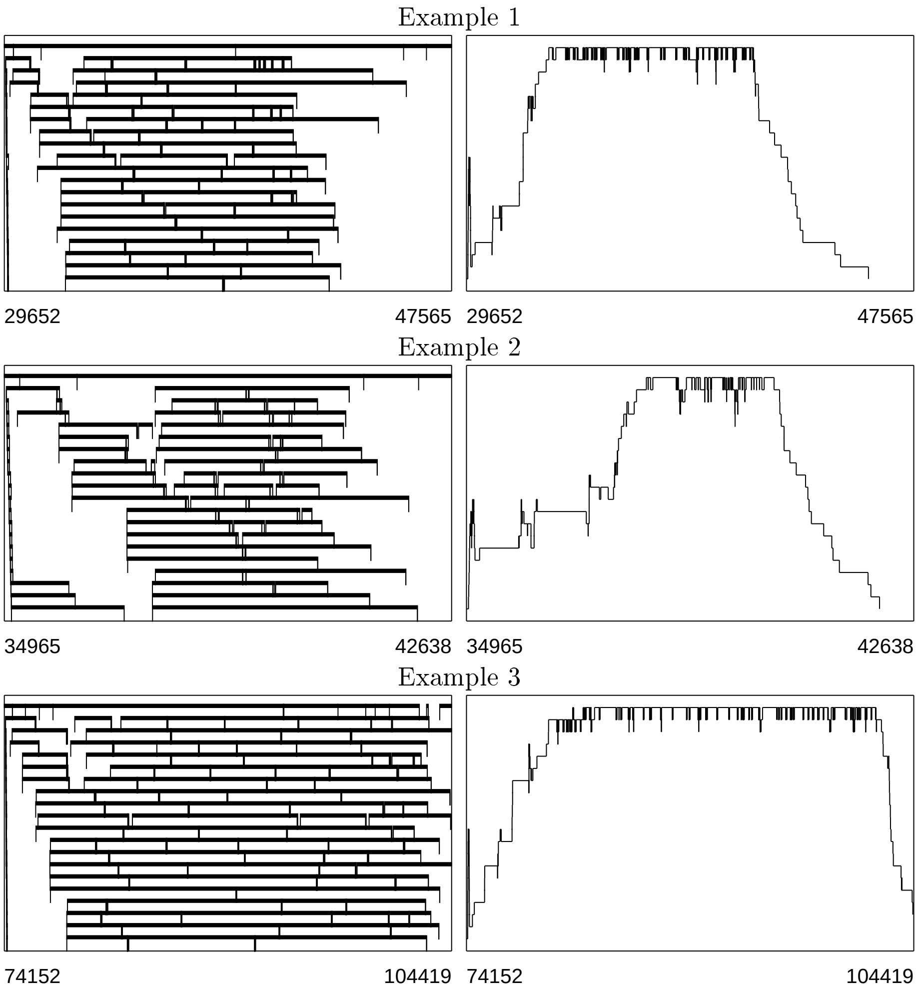


Figure 7: Dataflow Parallelism (Block Size 5)

3. more and more tasks get executed and utilization increases to maximum (task gap is filled),
4. many tasks are executed with maximum machine utilization (large block of tasks),
5. a few leftover tasks remain to be executed and the utilization drops to one (empty space on right boundary of diagram).

Especially Figure 7 shows that the frontend node (top line) has several tasks executed at the very end of the computation while no other nodes are active any more; this results from multiple tasks that were early scheduled to the frontend node and got blocked on data dependencies (which resulted in the creation of multiple Maple kernels on the frontend) such that these tasks had to be later resumed and completed.

Comparing these diagrams with those for the old algorithm in Figures 3 and 4 illustrate dramatic differences: In the new algorithm with $m = 4$, after the startup phase which takes 15–50% of the execution time, all 20 machines get saturated and compute tasks for 30–80% of the whole computation time. After that, the utilization curve drops rather sharply such that the final phase takes only 5-20% of the computation. The overall utilization rate is about 60–80%. In the $m = 5$ case, the startup phase yields earlier higher utilization rate; however, load imbalances in the later phase (caused by slower machines) let the utilization curve drop a bit earlier.

Execution Time The actual execution times of the new algorithm in comparison with the execution of the old algorithm are listed in Figure 8: the left row of diagrams shows the execution time of the new algorithm, the right row shows the speedup that the new algorithm gains *over the original one*. The comparison is based on four variants of the new algorithm labelled as follows:

- O: the original algorithm.
- N4, N5: the new algorithm with non-deterministic task selection in the manager program and in $\text{task}_{\text{Inner}}$ with block sizes $m = 4$ and $m = 5$.
- D4, D5: the new algorithm with deterministic task selection in the main program (first task in $tset$) and in $\text{task}_{\text{Inner}}$ ($t := tl$) with block sizes $m = 4$ and $m = 5$.

All variants of the new algorithm are considerably faster than the original version with an average speedup (compared to the original program) of 1.8 (Example 1) respectively 2.2 (Examples 2 and 3). We could thus reduce the sequential execution time from 552s, 198s, respectively 1798s on a PIII@450MHz PC down to a parallel execution time of 14s, 6s, respectively 29s. The fact that the speedups are drastically superlinear is caused by the changed order in which the greatest common divisors are computed (also in the original parallel algorithm) and is a hint for a general algorithmic improvement of the sequential algorithm.

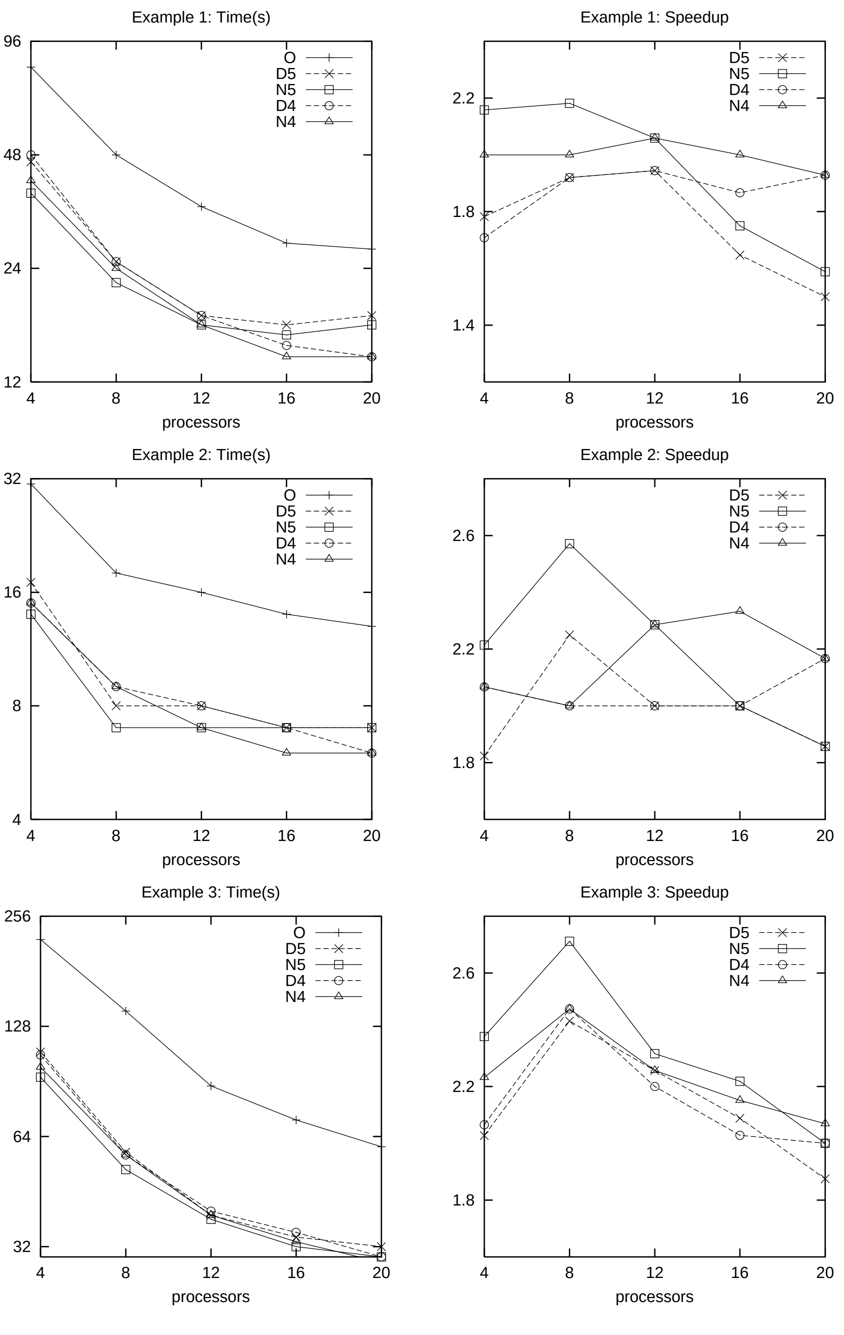


Figure 8: Execution Times

Block Size Comparing the algorithm variants N4 and N5, we see that for smaller processor numbers ($p \leq 12$) N5 is better while N4 seems to have some advantage for larger processor numbers. This result is consistent with our previous observation that larger block sizes may cause some load imbalance in the final phases of the computation (which is more significant for larger processor numbers). A similar trend can be seen when comparing D4 and D5.

Non-Determinism For analyzing the effect of non-determinism on the algorithm, we have counted for a sample run with 20 machines the number of times that the non-deterministic task synchronization operation `dist[select]` was called and how often it was not the first task that was selected because its result was not available (i.e., how often non-deterministic task selection made a difference compared to the deterministic variant): in the manager program, in 44% of all non-deterministic calls, it was not the first task that was selected; in $\text{task}_{\text{Inner}}$, this was the case in 19% of all calls. Thus there is a significant difference in synchronization dependencies, albeit two times more in the manager program than in the individual tasks.

As for analyzing the effect of non-determinism on the actual execution times, we compare the algorithm variants N5 with D5 and N4 with D4. We see that N5 is in many cases about 10% faster than N4, the application of non-deterministic brings a significant (but not dramatic) improvement in performance. In the comparison N4 versus D4, this effect is not that significant but still visible especially for smaller number of processors.

Speculation In all three examples, the amount of speculation (tasks whose result was not required) was not significant. The only tasks (i, j) that ever became superfluous where those with maximum $i + j$, i.e., those at the bottom of the triangular matrix (because other tasks in that row already yielded a gcd with constant degree).

Overall Comparison Figures 9, 10 and 11 illustrate the differences in the dynamic behaviors of D5, N5, D4 and N4 by traces of executions with 20 processors. The difference between non-deterministic and deterministic execution is visually captured by additional "gaps" in the traces of individual machines for the deterministic variant; the non-deterministic variant keeps after an initial phase processors continuously busy. The difference in block sizes becomes especially significant in the later phases of the algorithm where few late tasks may hamper the completion of the algorithm. The smaller the block sizes are, the better the overall utilization becomes also in the final phases of the computation.

6 Conclusions

We have rewritten a sample Distributed Maple application from a low-level imperative to a more high-level declarative style and analyzed the results in detail. This has demonstrated that in Distributed Maple the *dataflow* style of computing with implicit task synchronizations can considerably improve the

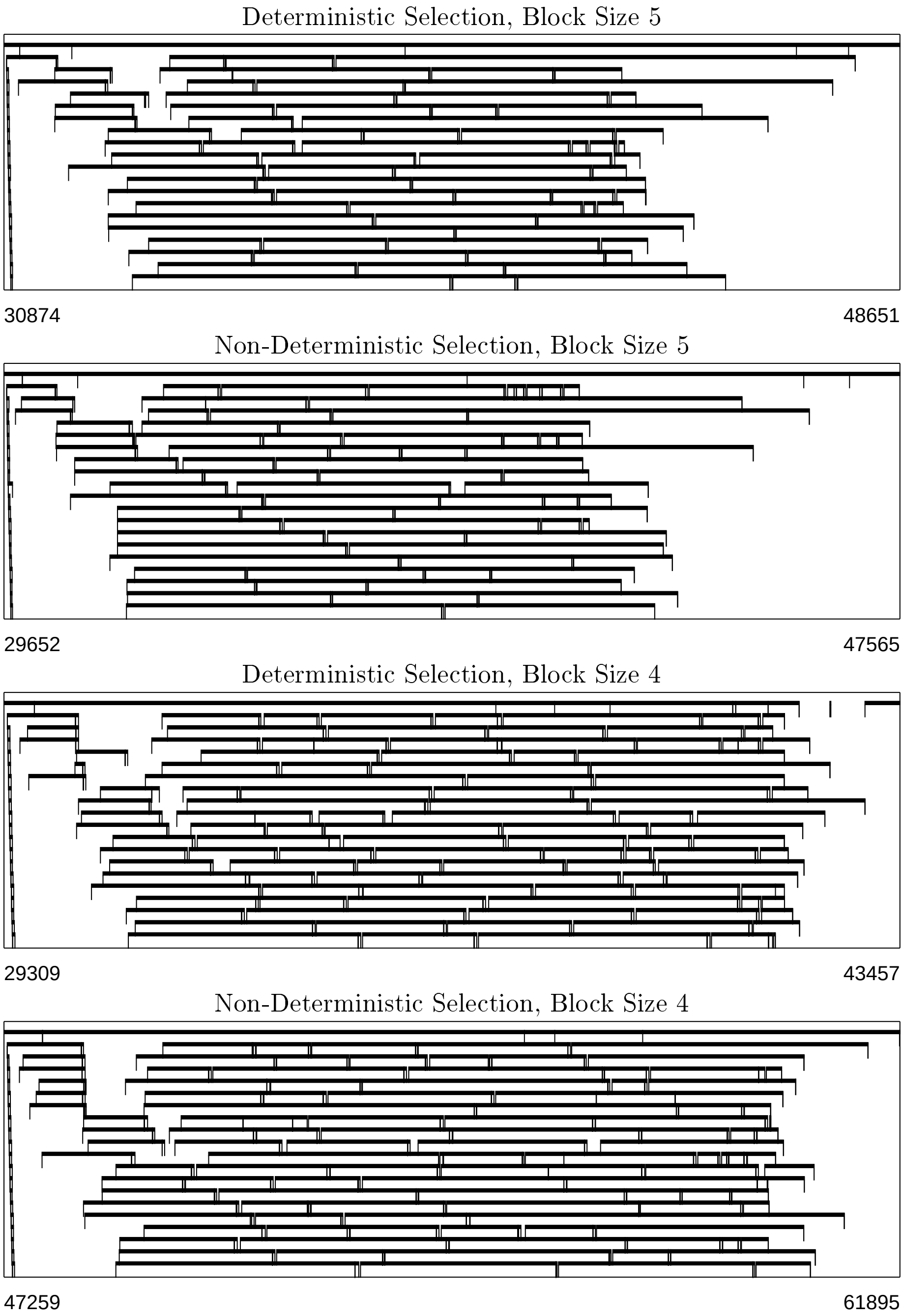


Figure 9: Example 1

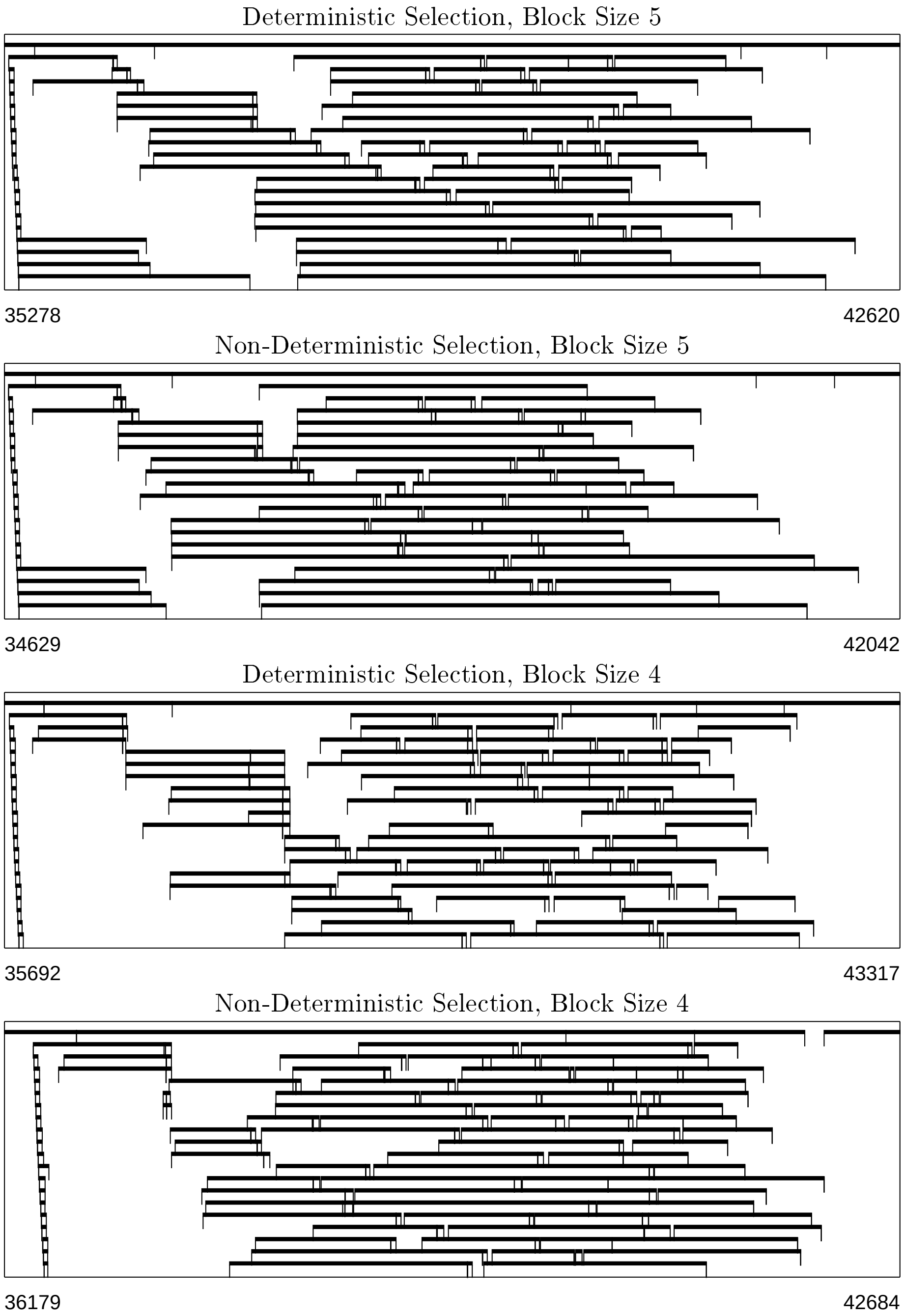


Figure 10: Example 2

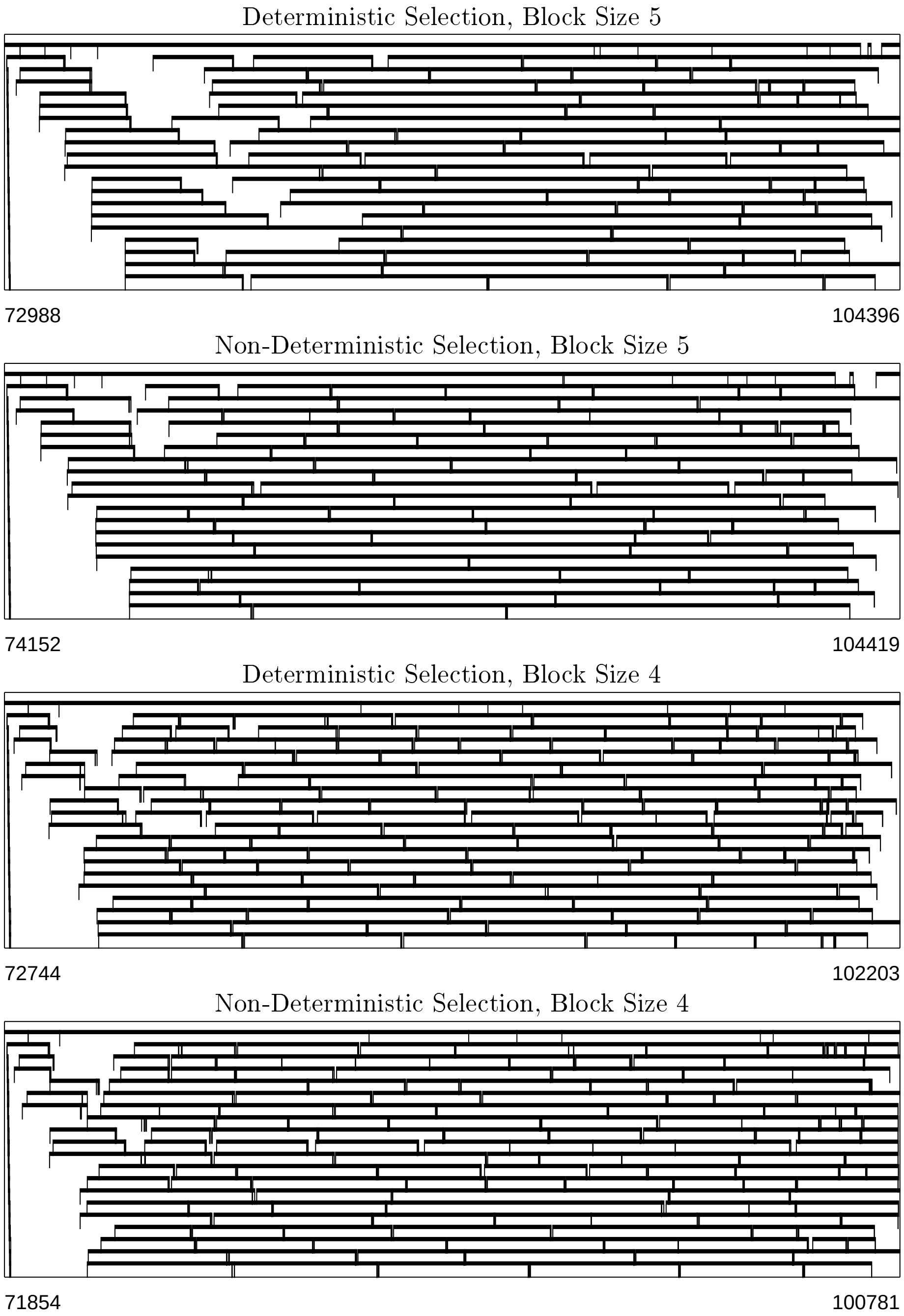


Figure 11: Example 3

performance of a parallel application: we have achieved a speedup of 2 compared to the original solution where the program explicitly schedules tasks for execution. While the original version is not able to saturate all processors with work, the dataflow solution yields a good utilization even for a higher number of processors.

We have also shown that a more declarative solution that uses *non-determinism* in order not to over-specify task dependencies may yield a *performance improvement of 10%* compared to a more imperative program that constrains data dependencies further than is algorithmically required.

Both results demonstrate that also in Distributed Maple a more declarative solutions that does not care about scheduling decisions may be more efficient than one where the programmer tries to keep explicit control over all aspects of the computation. Thus parallel declarative programs need not be a priori less efficient than parallel imperative ones. However, one aspect of declarativeness is non-determinism which should be also adequately addressed.

The presented work has been performed in preparation of a study that will compare parallel algorithms implemented in Distributed Maple with corresponding versions that will on the basis of a recently developed Haskell-Maple interface use the para-functional language Glasgow Parallel Haskell GPH for scheduling concurrent Maple computations.

References

[1] H.W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–751, 1999.

[2] Christian Mittermaier, Wolfgang Schreiner, and Franz Winkler. A Parallel Symbolic-Numerical Approach to Algebraic Curve Plotting. In Vladimir Gerdt and Ernst W. Mayr, editors, *CASC-2000, Third International Workshop on Computer Algebra in Scientific Computing*, Samarkand, Uzbekistan, October 5–9, 2000. Springer, Berlin.

[3] M. Mnuk and F. Winkler. CASA - A System for Computer Aided Constructive Algebraic Geometry. In J. Calmet and C. Limongelli, editors, *DISCO'96 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 1128 of *LNCS*, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.

[4] Wolfgang Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998. http://www.risc.uni-linz.ac.at/software/distmaple.

[5] Wolfgang Schreiner. Developing a Distributed System for Algebraic Geometry. In Barry H.V. Topping, editor, *EURO-CM-PAR'99 Third Euroconference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press, Edinburgh.

[6] Wolfgang Schreiner. Analyzing the Performance of Distributed Maple. Technical report, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, November 2000.

[7] Wolfgang Schreiner and Hans-Wolfgang Loidl. GHC-Maple Interface, November 2000. http://www.risc.uni-linz.ac.at/software/ghc-maple.

[8] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Analyzing Algebraic Curves by Cluster Computing. In Peter Kacsuk and Gabriele Kotsis, editors, *Distributed and Parallel Systems – From Instruction Parallelism to Cluster Computing, Proceedings of DAPSYS'2000, 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 175–184, Balatonfred, Lake Balaton, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.

[9] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismller, editors, *Euro-Par 2000, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.

[10] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Plotting Algebraic Space Curves by Cluster Computing. In Xiao-Shan Gao and Dongming Wang, editors, *ASCM'2000, 4th Asian Symposium on Computer Mathematics*, Chiang Mai, Thailand, December 17-21, 2000. World Scientific Publishers, Singapore/River Edge.