Manager-Worker Parallelism versus Dataflow in a Distributed Computer Algebra System*

Wolfgang Schreiner

Wolfgang.Schreiner@risc.uni-linz.ac.at
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Tel. +43 732 2468 9966, Fax. +43 732 2468 9930

Abstract. We analyze two implementation variants of a parallel computer algebra algorithm in Distributed Maple. The original solution uses a manager-worker mechanism to control task scheduling, which requires an elaborate administration scheme. The new algorithm is based on 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.

1 Introduction

We have developed Distributed Maple [8, 9] 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 computer algebra and algebraic geometry [12, 11, 6, 13]. The system is based on previous experience of other authors with the parallelization of Maple applications [14, 1, 2].

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.

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

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

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 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 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 [7] and was previously parallelized in a manager-worker style [11].

We have performed this work in preparation of a broader activity where we will compare parallel algorithms implemented in Distributed Maple with corresponding declarative versions written in para-functional language Glasgow Parallel Haskell GPH [3] which has been previously used for the parallelization of computer algebra algorithms [4]. We are going to use GPH as a coordination environment for scheduling computations among Maple kernels; for this purpose, we have already developed a GPH-Maple interface [10].

2 Distributed Maple

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple [9]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node of a session comprises two components (see Figure 1):

Scheduler The Java program `dist.Scheduler` coordinates the node interaction. The scheduler process attached to the frontend kernel starts instances of the scheduler on other machines and communicates with them via sockets.

Maple Interface The program `dist.maple` running on every Maple kernel implements the interface between kernel and scheduler. Both components use pipes to exchange messages (which may embed any Maple objects).

The user interacts with Distributed Maple via the Maple frontend by a number of programming commands, in particular:

`dist[start](`*f*`, `*a*`, ...)` creates a task evaluating *f(a, ...)* and returns a task reference *t*. Tasks may create other tasks and arbitrary Maple objects (including task references) may be passed as arguments and returned as results.

`dist[wait](`*t*`)` blocks the execution of the current task until the task represented by *t* has terminated and returns its result. Multiple tasks may independently wait for and retrieve the result of the same task *t*.

This model is based on para-functional principles which is sufficient for many kinds of computer algebra algorithms. The environment also supports nondeterministic task synchronization for speculative parallelism and self-synchronized shared data objects which allow tasks to communicate by a global store.

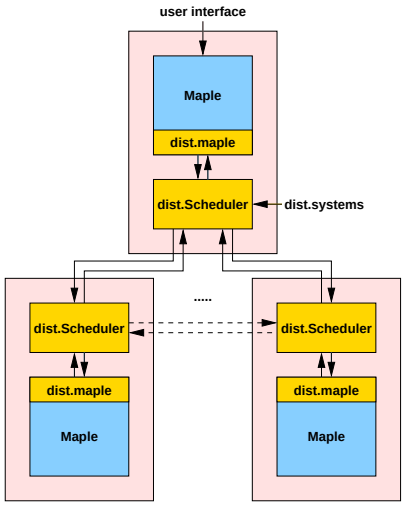

Fig. 1. Distributed Maple Architecture

3 Problem and Sequential Algorithm

Our problem can be summarized as follows [5]: 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 $\frac{\partial p^{u+v}}{\partial x^u \partial z^v}$ 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 shown in Figure 2:

(b, n) := derivatives(p)
  $i := 0$
  $d_0 := p$
  $b_0 := \mathrm{gsfd}(d_0(x,0))$
  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**
      $b_{i+1} := \gcd(b_{i+1}, d_j(x,0))$
    $i := i+1$
  $n := i$
end

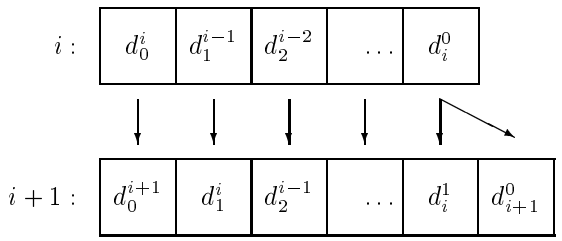

Fig. 2. Computation of Derivatives of Total Order $i+1$

4 Manager-Worker Parallelism

The main parallelization idea is to organize the computation of all d_u^v in a triangular matrix as shown in Figure 3: 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 at $(i-1,j)$ is available (if $i > j$) respectively the result at $(i-1,j-1)$ is available (if $i = j$).

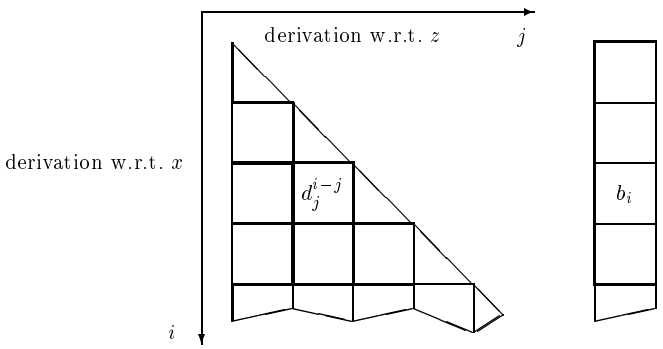

Fig. 3. The Matrix of Partial Derivations

In order to allow an efficient implementation, the algorithm must increase 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 has computed all those derivatives that represent the diagonal boundary of a triangular block, it starts a corresponding task that computes this block. Whenever a task has terminated, the manager 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 by the following *scheduling strategy*: Initially, the manager creates tasks for the first p triangular blocks. Whenever a task terminates, we "enable" the task that computes the adjacent square block; if a 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

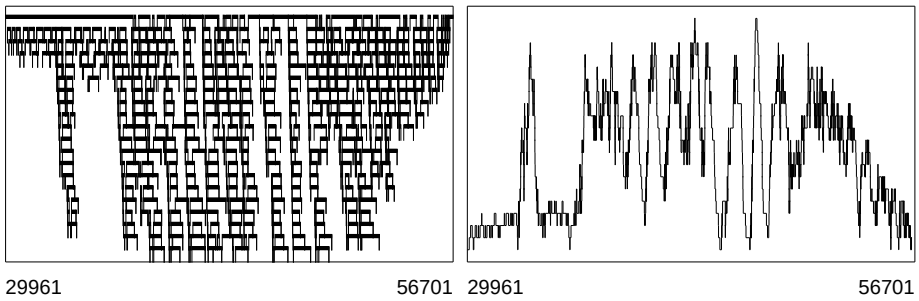

Fig. 4. Manager-Worker Parallelism

We have implemented this algorithm in Distributed Maple [11]. Figure 4 illustrates by one example 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 this particular example took on a PIII@450MHz PC 552s; the corresponding parallel computation took 27s.

While the parallelization achieved significant speedup (mostly 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 utilized. Apparently, the manager 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. It also does not help to increase the block size: while the manager is then no more the bottleneck, simply too few tasks are generated to saturate all processors and the overall execution time increases.

5 Dataflow Parallelism

We now explore a new parallel solution where tasks are implicitly scheduled for execution whenever their data dependencies have been resolved. Having recognized the utilization problem, we 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 5: we partition the matrix into

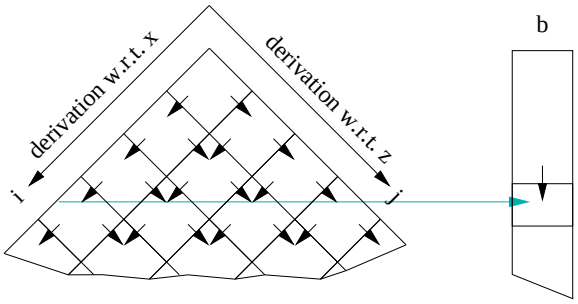

Fig. 5. Data Dependencies

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 compute within a block the gcd of all d_u^v with same sum $u + v$ which then contributes to the result vector b_{u+v}. Unlike in the original solution, the lines/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 depends on its unique upper neighbor $(0, j - m)$ (because it needs d_0^{j-1}) and a block $(i, 0)$ at the left boundary depends on its unique upper neighbor $(i - m, j)$ (because it needs d_{i-1}^j), a block (i, j) in the inner of the triangle depends on *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 $d_i^{n_z}$ 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 denote by

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

the set of all task positions (i, j) such that $i + j = k$.

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:

(b, n) := derivatives(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** task$_{\mathrm{R}}(i,\ j,\ m,\ t_{i,j-1})$
      else if $j = 0$ **then** $t_{i,j}$:= **start** task$_{\mathrm{L}}(i,\ j,\ m,\ t_{i-1,j})$
      else $t_{i,j}$:= **start** task$_{\mathrm{I}}(i,\ j,\ m,\ t_{i,j-1},\ t_{i-1,j})$
    end
  end
  for k **from** 0 **to** n **by** m **do**
    $tset := \{t_{i,j} : (i, j) \in \text{tasks}(k)\}$
    while $tset \neq \emptyset \wedge n = \min(n_x, n_z)$ **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** $\min(n_x, n_z)$ **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 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):

(p_1, p_2, g) := task(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:

```
(p1, p2, g) := taskL(i, j, m, t)
  ((p, _), _) := wait t
  (p1, p2, g) := task(i, j, m, ∂p/∂x)
end

(p1, p2, g) := taskR(i, j, m, t)
  ((_, p), _) := wait t
  (p1, p2, g) := task(i, j, m, ∂p/∂z)
end

(p1, p2, g) := taskI(i, j, m, tr, tl)
  t := select{tr, tl}
  if t = tr
    then (p1, p2, g) := taskR(i, j, m, t)
    else (p1, p2, g) := taskL(i, j, m, t)
  end
end
```

An "inner" task selects non-deterministically the result of one of the two neighbor tasks (whichever is available first) and, depending on the selection, proceeds like a task on the right boundary respectively on the left boundary.

Actually, a task (i, j) needs not return the boundary results 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 them (actually a task can do this ahead of the rest of its computation such that a subsequent task becomes enabled before its predecessor has terminated). Consequently, each task needs not receive references to other tasks but only the locations of shared data where it may find the boundary results. The main program creates these locations and passes to each task the location of its own result and the locations of the results it depends on. To avoid race conditions between producer and consumer, such a shared data item needs to be *self-synchronized*: as long as it is empty, the consumer gets blocked on an attempt to retrieve its content; the consumer is released only when the producer provides the content.

The main differences of the new algorithm to the original one are as follows:

1. **Matrix Decomposition:** the new algorithm uses a more regular decomposition. This has the advantage that each task computes the same number of derivatives m^2 the price that each task returns a gcd vector of double length. 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. It thus becomes much simpler than the original one where dependencies were explicitly maintained to schedule exactly one task per processor at every time. Thus the new algorithm should yield 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 larger memory requirements. Thus greater simplicity and better processor utilization at the price of larger memory usage are the main distinguishing features of the new solution.
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

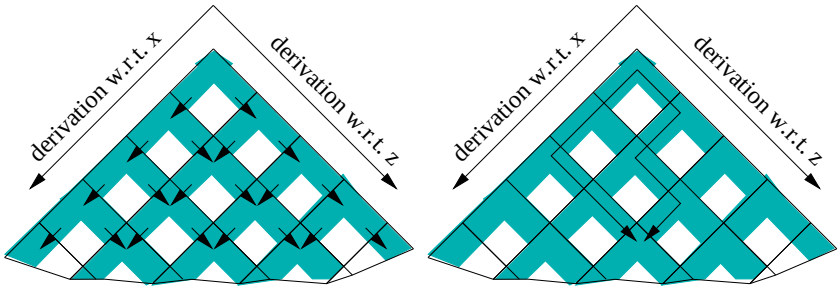

Fig. 6. Data Dependencies and Sample Dependency Paths

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 left diagram in Figure 6 denote those elements of the derivation matrix which are *not* required to enable any task to compute its results. Again, a similar optimization might have been performed in the original algorithm.
5. **Non-Determinism:** the new algorithm does not specify which of the two possible predecessor tasks returns the boundary polynomial that enables the further execution of an inner task. An inner task may thus be enabled by different paths of task dependencies as illustrated by the right diagram in Figure 6 that shows two possible paths that enable 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 . In the original algorithm, on each of the p machines at most one task was active of a time, such that at most $p-1$ tasks may become superfluous. In the new algorithm, about *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.

How far the speculated differences between original and new algorithm are actually relevant, is experimentally investigated in the following section.

6 Experimental Results

We have implemented in Distributed Maple the algorithm described in the previous section (including the optimization of using shared data spaces). The new implementation required about 140 lines of Maple code (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).

Behavior We have benchmarked the program in our local network with 20 Linux PCs with various Pentium class processors connected by switched 100MBit Ethernet lines; such a "Beowulf cluster" is currently the most suitable one for Distributed Maple applications. We have executed the sample benchmark problems as described in Section 4 and illustrate in Figures 7 two executions with block sizes $m = 4$ and $m = 5$ (which generally gave best results).

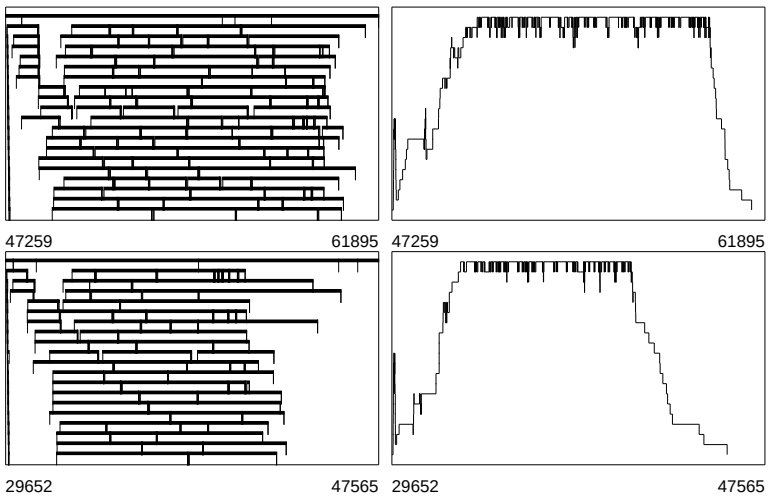

Fig. 7. Dataflow Parallelism (Block Sizes 4 and 5)

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);
3. more and more tasks get executed and utilization increases to maximum (task gap is filled),
4. many tasks are executed yielding maximum 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).

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

Comparing these diagrams with those for the old algorithm in Figure 4 illustrate dramatic differences: In the new algorithm with $m = 4$, after the startup phase which takes 20% of the execution time, all 20 machines get saturated and compute tasks for 65% of the whole computation time. After that, the utilization curve drops rather sharply such that the final phase takes only 15% of the computation. The overall utilization rate is about 75%. 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 earlier.

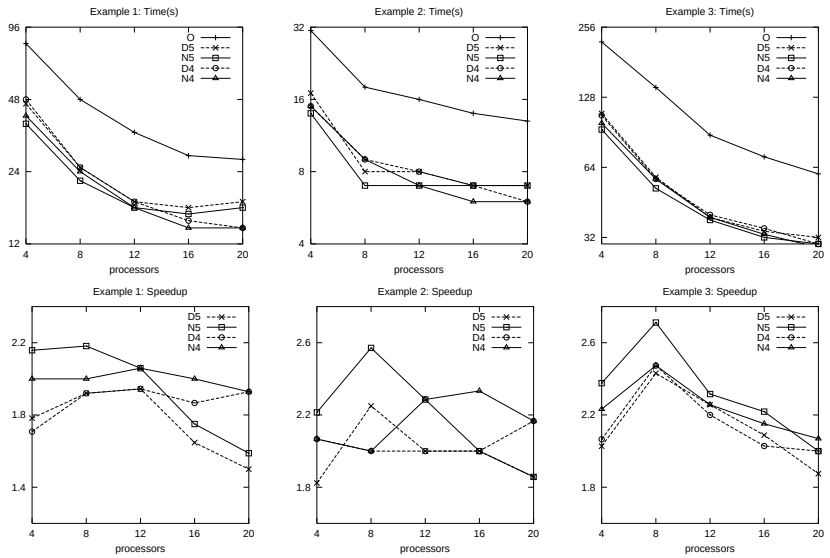

Fig. 8. Execution Times and Speedups

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 top row of diagrams shows the execution time of the programs, the bottom row shows the speedup that the new algorithm gains *over the original one*. The comparison is based on the algorithm variants labelled as follows:

- O: the original algorithm.
- N4, N5: the new algorithm with non-deterministic task selection in the main program and in task_I with block sizes $m = 4$ and $m = 5$.
- D4, D5: the new algorithm with deterministic selection in the main program ($t := \text{first}(tset)$) and in task_I ($t := tl$) 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 version) 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); this is a hint for a general algorithmic improvement of the sequential algorithm.

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 changes of task dependencies, we have counted for a sample run with 20 machines the number of times that the non-deterministic task synchronization operation 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 actually made a difference compared to the deterministic variant): in the main program, it was in 44% of all non-deterministic calls not the first task that was selected; in task_I, this was in 19% of all calls the case. Thus there is a significant difference in synchronization dependencies, albeit two times more in the main program than in the individual inner tasks.

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 D5, 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.

Figure 9 illustrate the differences in the dynamic behaviors of D5 and N5 by the trace of an execution 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 large 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.

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).

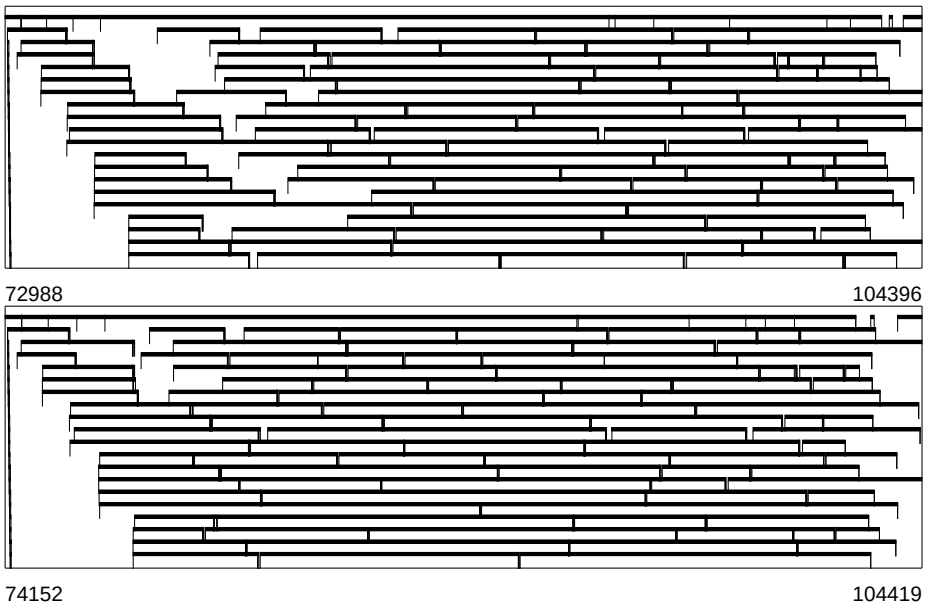

Fig. 9. Deterministic vs Non-Deterministic Selection

7 Conclusions

We have rewritten a Distributed Maple application from a low-level imperative to a more high-level declarative style and compared the results. This has demonstrated that the *dataflow* style of computing with implicit task synchronizations can considerably improve the performance: 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, the dataflow solution yields good utilization even for a larger processor number. We have also shown that a 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 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. The presented work has been performed in preparation of a study that will compare parallel algorithms implemented in Distributed Maple with corresponding declarative versions that will (on the basis of a recently developed Haskell-Maple interface) use the para-functional language Glasgow Parallel Haskell GPH.

References

[1] L. Bernadin. Maple on a Massively Parallel, Distributed Memory Machine. In *PASCO'97 — Second Int. Symp. on Parallel Symbolic Computation*, pages 217–222, Maui, Hawaii, July 20–22, 1997. ACM Press.

[2] A. Diaz and E. Kaltofen. FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation. In O. Gloor, editor, *ISSAC'98 Int. Symp. on Symbolic and Algebraic Comp.* ACM Press, New York, 1998.

[3] 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.

[4] H.W. Loidl. LinSolv: A Case Study in Strategic Parallelism. *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, 1997.

[5] C. Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Master's thesis, Johannes Kepler University, Linz, Austria, 2000.

[6] Christian Mittermaier, Wolfgang Schreiner, and Franz Winkler. A Parallel Symbolic-Numerical Approach to Algebraic Curve Plotting. In V. Gerdt and E. W. Mayr, editors, *CASC-2000, 3rd Int. Workshop on Computer Algebra in Scientific Computing*, Samarkand, Uzbekistan, October 5–9, 2000. Springer, Berlin.

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

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

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

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

[11] 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, Balatonfüred, Lake Balaton, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.

[12] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In A. Bode et al (eds.), *Euro-Par 2000*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.

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

[14] K. Siegl. Parallelizing Algorithms for Symbolic Computation Using ‖MAPLE‖. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press.