PERFORMANCE MODELLING FOR TASK-PARALLEL PROGRAMS

Matthias Kühnemann
Department of Computer Science
Chemnitz University of Technology
Germany

Thomas Rauber
Department of Mathematics and Physics
University of Bayreuth
Germany

Gudula Rünger
Department of Computer Science
Chemnitz University of Technology
Germany

Abstract Many applications from scientific computing and physical simulations can benefit from a mixed task and data parallel implementation on parallel machines with a distributed memory organization, but it may also be the case that a pure data parallel implementation leads to faster execution times. Since the effort for writing a mixed task and data parallel implementation is large, it would be useful to have an a priori estimation of the possible benefits of such an implementation on a given parallel machine. In this article, we propose an estimation method for the execution time that is based on the modelling of computation and communication times by runtime formulas. The effect of concurrent message transmissions is captured by a contention factor for the specific target machine. To demonstrate the usefulness of the approach, we consider a complex method for the solution of ordinary differential equations with a potential for a mixed task and data parallel execution. As distributed memory machine we consider the Cray T3E and a Linux cluster.

Keywords: execution time analysis, runtime formulas, mixed task and data parallelism, scientific computing.

1. Introduction

We consider task parallel programs in a form where the entire program is built up from multi-processor tasks (M-task) each of which can be executed on an arbitrary number of processors and is often implemented in a data-parallel way. Different M-tasks can cooperate in a compositional way which means that one task produces data to be used as input data by another task. But a program may also contain independent tasks which can be executed concurrently to each other on subsets of processors, also called groups of processors. There usually exists a large variety of different parallel realizations of the same application program. The program versions can differ in the execution order of the tasks and the mapping of each M-task onto a set of processors of a specific size. In this paper we address the question how to select an efficient parallel program version from the described class of programs when using a distributed memory machine (DMM).

Whether a pure data parallel or a mixed task and data parallel program version is more efficient strongly depends on the specific application program and its requirement for communication, especially for collective communication operations. Collective communication operations performed on smaller processor groups lead to smaller execution times due to the logarithmic or linear dependence of the communication times on the number of processors [14, 10, 7]. This behavior of the communication time can be an advantage for a concurrent computations on disjoint subsets of processors. In general, the effort to implement a task parallel program is considerable and so it would be useful to have an effective method to determine the most efficient program version before implementing all the details. Such a method needs a notion of costs assigned to different program versions under consideration where the costs of a specific version meet some requirements: they should be based on the specific task structure and they should reflect the behavior of the parallel execution times resulting on a specific DMM.

In this paper, we suggest a method based on costs estimating parallel execution times of task parallel programs with M-tasks. The modelling of costs is based on a specification of the task structure and assumes a corresponding partition of the set of processors into groups or subsets of processors. The costs are given as runtime formulas that contain parameters describing the parallel target machine and characteristics of the algorithm to be implemented [11]. Runtime formulas have been used before for modelling the execution time of communication operations in isolation [14, 10, 6], but their use for modelling the execution time of task parallel programs has not yet been studied. The runtime formulas are built up from computation costs and communication costs which reflect the actual execution of a computation or a communication operation in isolation. But due to complex processor architectures, memory hierarchies and network properties, the execution time of an entire program might strongly be influenced by cache effects or network contention. Our aim is to investigate whether a purely program-oriented construction of the runtime formulas is suitable to model costs for task parallel executions and whether additional effects like cache effects and network contention have to be taken into consideration to obtain an accurate prediction.

In the following sections, we describe how to build up program-specific runtime formulas for parallel platforms with distributed memory. In Section 1.2 we summarize the general approach. Section 1.3 briefly introduces the parallel target platforms used, a Cray T3E and a Beowulf cluster of PCs, and presents the runtime formulas for communication operations for those machines. As complex example applications we consider one-step methods for solving ordinary differentiation equations (ODEs) which have a potential for M-task parallelism. In Section 1.4, we model the runtime formula for an entire application. Section 1.5 shows that the runtime formulas reflect the performance behavior of the parallel program and Section 1.6 concludes.

2. Runtime formulas

The runtime formulas of our cost model are composed according to the M-task structure of the programs whose execution time they described in [11]and summarized in the following. The runtime formulas consist of two parts describing the computation times and the communication times.

Computation and communication operations. The computation times are modelled by taking into account the number of operations to be executed. For each arithmetic operation we use a computation time t_{op} that can be determined by runtime measurements with simple sequential test programs on the parallel machine used. In addition, we use execution times T_f for specific functions f which are needed to describe specific application problems. The execution time for such a function f can be build up from single computation operations or can be a measured time for the function in isolation. The reason to use a single value T_f in the runtime formula is that most numerical methods, like ODE solvers, are designed as black-box solvers, so that they can be used for arbitrary functions f describing the right-hand side of the ODE system.

The communication times are modelled by formulas that describe the execution time of individual communication operations, such as single-transfer or broadcast operations. These formulas are used to model the internal communication time of a M-task, which are often realized in a data parallel way. Communication formulas may also be used to describe the time for data redistributions inserted between different multi-processor tasks related in a compositional way. The runtime formulas for communication operations are functions in closed form that depend on the message size b and the number p of processors participating in the communication operation. For a given DMM, the value of these parameters result from modelling the runtime formulas by the least squares method. In Section 1.3 we present runtime formulas for different communication operations in isolation, see also [14, 10, 6, 4].

Task parallel programs. Runtime formulas for entire programs are built up according to the task structure of the specific program. First, each participating M-task M is assigned a runtime formula built up from communication and computation times according to the internal structure. We assume a data parallel or SPMD-like internal computation structure consisting of alternating phases of computation and communication, so that the execution time can be represented by

$$T_M(p,b) = T_{comp}(p,b) + T_{comm}(p,b).$$

$T_{comp}(p,b)$ is the maximum of the execution times of the participating processors. $T_{comm}(p,b)$ is the sum of the runtimes of the communication operations used. Second, the runtime of an entire program is built up from the costs of the M-tasks according to the task structure. In the case that two M-tasks M_1 and M_2 are executed concurrently to each other on disjoint sets of processors, the maximum of the runtimes is taken:

$$max(T_{M_1}(p_1,n_1), T_{M_2}(p_2,n_2)).$$

$T_{M_i}(p_i,n_i)$ denotes the runtime of task M_i for problem size n_i on a group of p_i processors, i=1,2. The sets of processors executing M_1 and M_2 are assumed to be disjoint. In the case that two M-tasks M_1 and M_2 are executed one after another and the runtimes are added:

$$(T_{M_1}(p,n_1) + T_{M_2}(p,n_2)) + T_{redist}(p,m).$$

Both times $T_{M_i}(p,n_i)$ contain the same parameter p for the sizes of the processor groups indicating that both tasks M_1 and M_2 are executed on a processor group of the same size, which is usually the same group. In addition, some communication between successive task activations might be necessary to realize redistribution of data. The time needed for the redistribution is denoted by $T_{redist}(p,m)$ where p is the number of participating processors and m the size of the data to be redistributed.

2.1 Comparison with other cost models

Our approach of performance modelling with runtime formulas combines a modelling part for SPMD-like computations and a construction part for the upper level task parallelism of M-tasks. The combination is inspired by the specific goal to compare the efficiency of different realizations with mixed task and data parallelism for the same given algorithm. Other cost models for parallel programming, like the LogP model and its variations [2, 1]or the BSP model [8, 5], are aimed at aspects different from ours. The LogP model is more architecture oriented with an emphasis on a detailed modelling of parallel runtimes but is less suited to express explicitly upper level task parallelism. The BSP model concentrates on a specific program structure of supersteps which is not straightforwardly related to a mixed task and data parallelism with M-tasks. The advantage of using runtime formulas for the modelling of the execution times lies in the fact that they can easily be adapted to a mixed task and data parallel execution of program parts. Moreover, runtime formulas are able to capture the effect that the same communication operation may lead to different execution times, depending on whether other communication operations are simultaneously executed by concurrent processor groups or not. This effect cannot easily be captured by other models like the BSP model [8, 5]or the LogP model and its variations [2, 1]that assume a pure data-parallel execution of the program. In these models, it is difficult to capture runtime effects that occur only in a task parallel execution. Using runtime formulas, such effects can be captured, e.g., by contention factors introduced in Section 1.3.2.

3. Modelling Communication Costs

We consider the communication on different machine with distributed memory, a Cray T3E and a Beowulf-Cluster. The T3E uses a three-dimensional torus network. The six communication links of each node are able to simultaneously support hardware transfer rates of 600 MB/s for the T3E.
The Beowulf Cluster CLiC ('**C**hemnitzer **Li**nux **C**luster') is build up of 528 Pentium III processors clocked at 800 MHz. The processors are connected by two different networks, the communication network and the service network. Both are based on the fast-Ethernet-standard, i.e., the PEs can swap 100 MBit per second. The service network (Extreme Block Diamand) allows external access to the cluster. The communication network (Cisco Catalyst) is used for inter-process communication between the PEs.

We consider message-passing programs that are coded using the MPI standard [3]. On the CLiC, LAM MPI 6.3.2 is used.

3.1 Communication operations in isolation

For single-transfer operations and collective communication operations we consider different variants. The message sizes are between 2 KByte and 400 KByte. Except for some anomalies of the buffered single-to-single transfer on the Cray T3E, the runtimes increase linearly with the message size. On the Cray T3E-1200 the standard single-to-single transfer is the fastest operation. For the CLiC we have additionally implemented a piecewise single-to-single transfer operation that splits the message to be transmitted into pieces of 4KB and sends the pieces separately. This is the fastest single-transfer operation on the CLiC.

For collective communication operations we consider single-broadcast operations (`MPI_Bcast()`), accumulation operations (`MPI_Reduce()`), gather operations (`MPI_Gather()`), scatter operations (`MPI_Scatter()`), and multi-broadcast operations (`MPI_Allgather()`). The execution times for collective communication operations on the Cray are essentially faster than on the CLiC. Again, we have implemented piecewise communication operations on the CLiC that turn out to be faster than the original operations.

Runtime formulas: The runtime formulas that are used for the modelling are summarized in Table 1. The value b denotes the message size in bytes, p is the number of processors participating in the communication operation. The coefficients τ and t_c can be considered as startup and byte-transfer times and are determined by curve fitting with the least-squares method. For different internal realization of the same communication operation, different values for the coefficients are obtained. We therefore use an additional parameter V to distinguish the different variants; the specific values for the coefficients τ (V), t_c(V), t_1(V) and t_2(V) are given in Tables 2 - 3.

Table 1. Runtime formulas for communication operations.

operation	*runtime formula*
MPI_Send	$t_{s2s}(b) = \tau + t_c \cdot b$
MPI_Bcast	$t_{sb}(p,b) = \tau log_2(p) + t_c \cdot log_2(p) \cdot b$
MPI_Reduce	$t_{sa}(p,b) = \tau log_2(p) + t_c \cdot log_2(p) \cdot b$
MPI_Allgather	$t_{mb}(p,b) = \tau_1 + \tau_2 \cdot p + t_c \cdot p \cdot b$

Single transfer: The runtime for a single-to-single transfer is modelled by a linear function $t_{s2s}(b) = \tau + t_c \cdot b$ where $\tau(V)$ denotes a *startup time* and $t_c(V)$ denotes the *byte-transfer time* for the specific operation V. Curve fitting results in the parameter values given in Table 2. The negative coefficients arise when using the least squares method for measured runtimes in the complete range of message sizes. Restricting the method to small message sizes up to 2 KByte leads to a separate, more accurate runtime formula for small messages without negative coefficients. A comparison between the predicted and measured runtimes for all MPI single-transfer operations shows that the predictions fit the measured runtimes quite accurately, especially for large messages (not shown in a figure).

Single broadcast and accumulation: The modelling of single-broadcast (`MPI_Bcast()`) operations, see Table 1, uses a logarithmic dependence on the number p of processors because the broadcast transmissions are based on

Table 2. Parameter values for single-transfer operations and for the runtime of collective operations with a logarithmic dependence on the number of participating processors.

Coefficient for runtime formula of single-transfer, single-broadcast and accumulation				
	CLiC		Cray T3E	
variant	$\tau(V)[\mu s]$	$t_c(V)[\mu s]$	$\tau(V)[\mu s]$	$t_c(V)[\mu s]$
MPI_Send	-7.526	0.10607	13.965	0.00267
Send_P	145.021	0.08804	9.748	0.00533
MPI_Bcast	-3420.688	0.3409	7.723	0.0039
Bcast_P	564.125	0.0939	7.007	0.0050
MPI_Reduce	-119.871	0.1223	168.516	0.0093

broadcast trees with logarithmic depth. The same formula can also be used for the prediction of single-accumulation operations (`MPI_Reduce()`).

Multi-broadcast and gather/scatter: The runtime formula for multi-broadcast operations (`MPI_Allgather()`), see Table 1, increases linearly with the message size and the number of processors. This formula can also be used to model scatter and gather operations. The resulting coefficients are shown in Table 3.

Table 3. Parameter values for the runtime formulas with a linear dependence on the number of participating processors. MultiBcast_P and Allgather_P are piecewise implementations.

Coefficient for multi-broadcast/gather/scatter						
	CLiC $[\mu s]$			Cray T3E $[\mu s]$		
variant	$\tau_1(V)$	$\tau_2(V)$	$t_c(V)$	$\tau_1(V)$	$\tau_2(V)$	$t_c(V)$
MultiBcast	-33270.8	21801.5	1.031	-3.72	42.60	0.028
MultiBcast_P	-9724.2	8385.5	0.690	-24.20	-0.81	0.036
MPI_Allgather	9175.3	-7542.0	3.182	6.04	-0.75	0.019
Allgather_P	-669.9	543.3	2.806	7.72	13.23	0.018
MPI_AllgatherV	-709.6	-957.0	5.443	11.83	17.47	0.019
MPI_Gather	-316.2	654.5	0.095	31.08	-118.51	0.006
MPI_Scatter	24.6	1439.5	0.086	-0.48	5.45	0.003

3.2 Communication in task parallel executions

In the last subsection, we have considered runtime formulas for the execution time of communication operations in isolation, which means that there was no concurrent communication operation of the same application program in transmission. In Section 1.4 those runtime formulas are used for the modelling and prediction of data parallel realizations of regular programs from scientific computing [10, 12]. But it is not a priori clear whether those runtime formulas can also be used for modelling the execution time of mixed task and data parallel programs and in fact, experiments have shown that using the runtime formulas from Subsection 1.3.1 leads to predicted runtimes that are too small compared to the measured runtimes. The effect is much larger on the Beowulf cluster than on the T3E. This behavior indicates that concurrent communication operations of the same application program can interfere with each other.

Further experiments have shown that only a slight change of the runtime formula for communication operations is needed to model the runtime of concurrent communication transmissions. The change can be concluded from the following observation made when using the runtime formulas for data parallel realizations from Subsection 1.3.1 to model the runtime for concurrent collective communication operations: The difference between the measured and the predicted runtimes is increasing with the number p of processors (used for the entire application program) and the size n of the message transmitted. The dependence on p is more significant than the dependence on n. Both dependencies correspond to the expectation that collisions in the network become more likely when the number of participating processors and the message sizes are increasing. To capture the interference of a concurrent execution of communication operations in the runtime formulas, we adjust the byte transfer time of the runtime formulas by introducing a network contention factor $C(p, n)$ that depends on p and n. In principle, this factor can be determined by a detailed queuing analysis, but this would require the knowledge of the internal realizations of the MPI communication operations. Thus, we take the approach to determine the contention factor empirically starting from the runtime estimation with the runtime formulas from Subsection 1.3.1.

We determine the contention factor by comparing the delay in the execution times for communication operations, when they are performed concurrently, with the execution times of these operations without network contention. The contention factor itself is modelled as a function of p and n and the shape of the contention factor (or contention function) has been determined by experiments from the measured delays in the execution times. The coefficients within this function are then determined by curve fitting. The resulting contention factor is used for the modelling of the communication times of those program parts of complex application programs that are executed in a mixed task and data parallel way. For each parallel machine, a different contention factor results because of the different network architectures of the machines. To summarize, using the contention factor, the structure of the runtime formulas for one communication operation does not change but the contention factor is used to adjust the byte-transfer time t_c to the specific situation. Especially, the startup times are not changed. Thus, for example, the runtime formula of a multi-broadcast operation is

$$\tilde{t_{mb}}(p,b) = \tau_1 + \tau_2 \cdot p + C(p,n) \cdot t_c \cdot p \cdot b. \qquad (1)$$

The same contention factor is used for each communication operation, i.e., there is no need to determine different contention factors for different communication operations. The communication operations of pure data parallel parts of application programs are modelled without the contention factor because there is no interference of message transmission. The contention factors for the modelling of concurrent message transmissions are

$$\begin{aligned} C_{T3E}(p,n) &= 0.04 \cdot p \cdot log_2(log_2(p)) \cdot log_2(n) \\ C_{CLiC}(p,n) &= 0.0045 \cdot p \cdot p \cdot log_2(p) \cdot (log_2(n) + log_2(p)), \end{aligned}$$

where p denotes the total number of processors participating in the execution and n the size of the message transmitted.

4. Example Application

To investigate the usefulness of the modelling approach for complex application programs, we consider parallel implementations of a specific solution method for ordinary differential equations (ODEs), the iterated Runge-Kutta method (iterated RK method).

4.1 Task structure of iterated RK methods

The iterated RK method is an explicit one-step method for the solution of initial value problems of ODEs. The iterated RK methods determines a sequence of approximation values y_1, y_2, y_3... for the exact solution of the ODE system in a series of sequential time steps. In each of the time steps a fixed number s of stage vectors are iteratively computed and combined to the next approximation vector in the following way:

```
for $l = 1, ..., s$ initialize stage vector $v^l_{(0)}$
  for $j = 1, ..., m$
    for $l = 1, ..., s$: compute new stage vector approximation $v^l_{(j)}$
compute new approximation vector $y^{(m)}_{k+1}$,
compute approximation vector $y^{(m-1)}_{k+1}$ for step size control.
```

The number m of iterations is given by the specific RK method. Each computation of a stage vector approximation requires an evaluation of the function f that describes the ODE to be solved. The advantage of the iterated RK methods for a parallel execution is that the iteration system of size $s \cdot n$ consists of s independent function evaluations that can be performed in parallel [9]. For systems of differential equations, an additional data parallelism can be exploited. Thus, the algorithm provides several possibilities for a parallel implementation. The computation of the stage vectors in one iteration j of the stage-vector computation can be performed on subsets of processors at the same time (*group implementation*) or alternatively by all processors one after another (*consecutive implementation*). The group implementation is a mixed task and data parallel implementation whereas the consecutive implementation is a pure data parallel realization.

4.2 Consecutive implementation

The computation time of the consecutive execution on p processors without the stepsize control can be modelled by the formula

$$T_{dp}(n,p) = \left(ms\left\lceil\frac{n}{p}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(ms\left\lceil\frac{n}{p}\right\rceil + s\left\lceil\frac{n}{p}\right\rceil\right)T_f + n\,s\,t_{op}$$

where T_f denotes the time for the evaluation of *one* component of f and t_{op} denotes the time for the execution of one arithmetic operation. In each loop body each processor has to compute n/p components of the argument vector using $2s+1$ operations and n/p components of f. Since f and its access pattern are not known in advance, the complete argument vector has to be made available to each processor with a multi-broadcast operation. For the communication time, the following formula is used with t_{mb} from Table 1.

$$C_{dp}(n,p) = s\,m\,t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right) + t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right).$$

4.3 Group implementation

A group implementation of the iterated RK method uses s independent groups of processors where each group G_i of size g_i, $i = 1, \ldots, s$, is responsible for computing the approximations of one specific stage vector.

The processor groups should be of equal size, since the computation of each stage vector approximation requires an equal amount of computation. As it is possible that p is not a multiple of s, the group with the smallest number $g_{min} = \lfloor p/s \rfloor$ of processors determines the computation time

$$\begin{aligned} T_{gp}(n,p) = {} & \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \\ & \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{g_{min}}\right\rceil\right)T_f + \left\lceil\frac{n}{g_{min}}\right\rceil\,s\,t_{op}. \end{aligned}$$

The communication time is modelled by the runtime formula with the machine specific contention factor, see Equation (1), to reflect the concurrently executed multi-broadcast operations.

$$C_{gp}(n,p) = 2\cdot m\cdot \tilde{t_{mb}}\left(g_{min}, \left\lceil\frac{n}{g_{min}}\right\rceil\right) + t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right).$$

The following pseudo-code illustrates the structure of the program.

```
forall $l \in \{1, \ldots, s\}$ do in parallel ( *group parallelism*)
    forall processors $q \in G_l$ do ( *data parallelism*) {
        compute $\lceil n/g_l \rceil$ components of $f(y_\kappa)$;
        initialize $\lceil n/g_l \rceil$ components of $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$;
    }
for $j = 1, \ldots, m$ do ( *sequential iteration*) {
    forall $l \in \{1, \ldots, s\}$ do in parallel ( *group parallelism*)
        forall $q \in G_l$ do ( *data parallelism inside groups*) {
            compute $\lceil n/g_l \rceil$ components of argument $\mu(l, j)$
            group-multi-broadcast of local components of $\mu(l, j)$;
            evaluate $\lceil n/g_l \rceil$ components of $1f(\mu(l, j))$;
            group-multi-broadcast of local components of $f(\mu(l, j))$
        }
forall processors $q$ do ( *data parallelism on all processors*)
    compute $\lceil n/p \rceil$ components of $y_{\kappa+1}$;
    multi-broadcast the computed components of $y_{\kappa+1}$;
}
stepsize control;
```

5. Runtime experiments

To validate the runtime formulas we have performed runtime tests on the and T3E and the CLiC for up to 128 processors. Since the execution time of the iterated RK method strongly depends on the ODE system to be solved, we consider two classes of ODE systems:

Sparse ODE systems. These ODE systems have a right-hand side function f for which the evaluation of each component has a fixed evaluation time that is independent of the size n of the ODE system (sparse function), i.e., the evaluation time of the entire function f consisting of n components increases linearly with the size of the ODE system.

Dense ODE systems. These ODE systems have a right-hand side function f for which the evaluation of each component has an evaluation time that increases linearly with n, i.e., the evaluation time of the entire function f increases quadratically with the size of the ODE system.

Dense ODE systems lead to the fact that the computation time usually dominates the communication time, i.e., the comparison of measured and predicted execution times shows how good the computation times are modelled. On the other hand, for sparse ODE systems the communication time plays an important role and the comparison also includes this part. We have used an iterated RK method with $s = 4$ stages that is based on an implicit RadauIIA method. This method leads to a convergence order of 7, if $m = 6$ iterations are executed in each time step. The runtimes shown in the following tables and figures are runtimes for one time step of the method and are obtained by averaging over a large number of time steps. The (measured and predicted) runtimes include the time for stepsize and error control.

On the T3E, the runtime of the *consecutive* implementations of the iterated RK method for *dense* ODE systems can be modelled very accurately and are not shown here. But sparse ODE systems also lead to good predictions. In this case, no concurrent message transmissions take place, and therefore the contention factor does not need to be used. For both cases, the predictions are quite accurate, but not as accurate as the predictions for dense ODE systems. Nevertheless, they are accurate enough to be used for predicting the effect of a task parallel implementation.

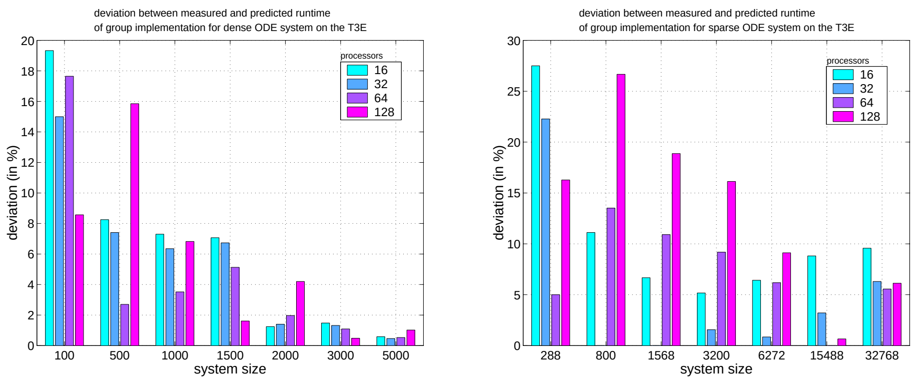

Figure 1. Deviation between measured and predicted runtime of group implementation for dense (left) and sparse (right) ODE system on the Cray T3E-1200.

Figure 1 shows the deviations between measured and predicted runtime for a *group* implementation of the iterated RK method for *dense* (left) and for *sparse* (right) ODE systems on a T3E, again using the contention factor for the concurrent message transmissions because of a task parallel execution. Figure 2 illustrates the accuracy of the *group* implementation for *dense* (left) and *sparse* (right) ODE systems of large sizes on the T3E for different numbers of processors. In contrast to dense ODE systems, the solution of sparse ODE systems leads only to considerable speedups for processor numbers of up to 16. For larger numbers of processors, the communication time dominates the computation times.

Table 4 compares measured and predicted execution times for *group* implementations of the iterated RK methods for *dense* ODE systems on the CLiC. Figure 3 shows an illustration for a fixed message size and different numbers of processors. For this machine, the deviations between the measured and pre-

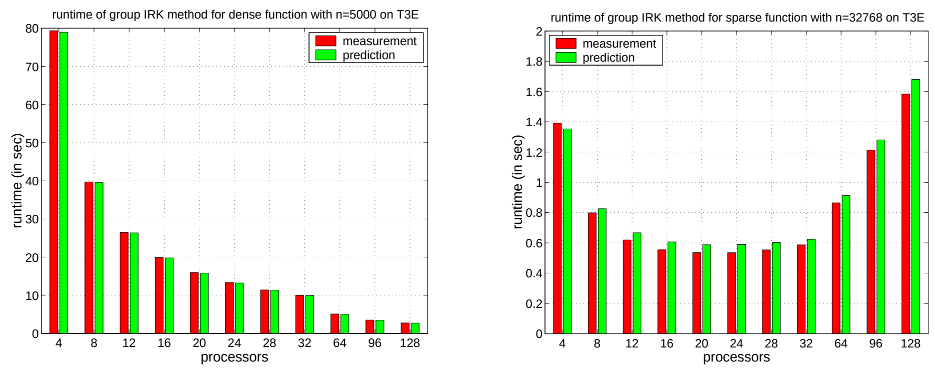

Figure 2. Measured and predicted execution times for a group implementation of one time step of the iterated RK method for a dense function of size $n = 5000$ (left) and for a Brusselator system of size $n = 32768$ (right) on a Cray T3E-1200.

Table 4. Group implementation for *dense* ODE system on the CLiC. The average deviation is 25.5 %.

	measured runtimes				*predicted runtimes*			
n	p=16	p=32	p=64	p=128	p=16	p=32	p=64	p=128
100	0.018	0.030	0.060	0.494	0.011	0.022	0.082	0.424
500	0.104	0.129	0.216	2.423	0.092	0.127	0.406	1.797
1000	0.301	0.296	1.288	3.425	0.283	0.306	0.840	3.667
1500	0.600	0.515	2.274	3.446	0.572	0.535	1.302	5.451
2000	1.049	1.492	2.418	4.296	0.959	0.813	1.790	7.372
3000	2.313	2.656	2.506	3.601	2.033	1.519	2.856	11.154
5000	7.814	5.180	4.714	15.303	5.361	3.528	5.289	19.036

dicted execution times are much larger than for the T3E, especially for a larger number of processors. Although the deviations may be quite large, they can still be used as a rough estimate of the performance of a task parallel execution. The main reason for the large deviations are caused by the large increase of the communication time with an increasing number of processors. But also for a smaller number of processors, there are considerable deviations for large system sizes because of caching effects which are caused by the memory hierarchy of the single processors (Intel Pentium III). Such effects are much smaller on the Alpha 21164 processor of the T3E-1200 because of its different cache organization, see [13]for a more detailed investigation of such effects. The runtimes on the CLiC show that even for dense ODE systems, the machine only leads to satisfactory speedups for up to 32 processors. For larger processor numbers, the communication time and the network contention are too high.

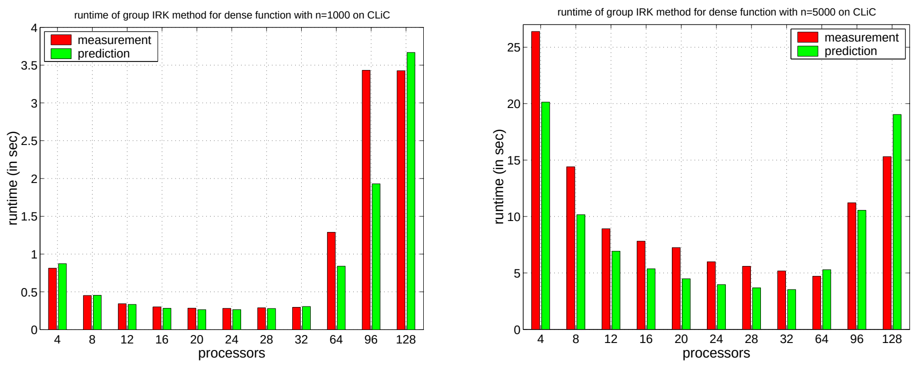

Figure 3. Measured and predicted execution times for a group implementation of one time step of the iterated RK method for dense ODE of size $n = 1000$ (left) and $n = 5000$ (right) on the CLiC Beowulf cluster.

6. Conclusions

In this article, we have shown that it is possible to model the execution times of mixed task and data parallel implementations by runtime formulas and that the use of a simple contention factor is sufficient to capture the interference of concurrent message transmissions. The runtime formulas model the execution times quite accurately for parallel machines like the T3E with a high-speed interconnection network. For a Beowulf cluster with an Ethernet-based network, the network contention caused by concurrent transmissions is much larger and it is more difficult to capture the effects by a simple contention factor. But the predictions are still reasonable and give a first impression of the possible effects of a task parallel realization.

References

[1] A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. Technical Report TRCS95-09, University of California at Santa Barbara, 1995.

[2] D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *4th Symp. on Principles and Practice of Parallel Programming*, 28(4):1–12, 1993.

[3] The MPI Forum. MPI: A Message Passing Interface Standard. Technical report, University Tennessee, April 1994.

[4] R. Foschia, T. Rauber, and G. Rünger. Modeling the Communication Behavior of the Intel Paragon. In *Proc. 5th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE, pages 117–124, 1997.

[5] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[6] K. Hwang, Z. Xu, and M. Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–536, 1996.

[7] S. Johnsson. Performance Modeling of Distributed Memory Architecture. *Journal of Parallel and Distributed Computing*, 12:300–312, 1991.

[8] W.F. McColl. Universal Computing. In *Proceedings of the EuroPar'96*, Springer LNCS 1123, pages 25–36, 1996.

[9] T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[10] T. Rauber and G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In *Proc. 11th Symp. on High Performance Computing Systems (HPCS'97)*, 1997.

[11] T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.

[12] T. Rauber and G. Rünger. Modelling the runtime of scientific programs on parallel computers. In *Proc. ICPP-Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-00)*, pages 307–314, Toronto, Kanada, August 2000.

[13] T. Rauber and G. Rünger. Optimizing Locality for ODE Solvers. In *Proc. of the 15th ACM Int. Conf. on Supercomputing*, pages 123–132. ACM Press, 2001.

[14] Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.