View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Juelich Shared Electronic Resources

John von Neumann Institute for Computing

A Framework for Performance-Aware Composition of Explicitly Parallel Components

Christoph W. Kessler, Welf Löwe

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 227-234, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

A Framework for Performance-Aware Composition of Explicitly Parallel Components

Christoph W. Kessler[1] **and Welf Löwe**[2]

[1] PELAB, IDA, Linköpings universitet, S-58183 Linköping, Sweden
E-mail: chrke@ida.liu.se

[2] MSI, Växjö universitet, Växjö, Sweden
E-mail: welf.lowe@msi.vxu.se

We describe the principles of a novel framework for performance-aware composition of explicitly parallel software components with implementation variants. Automatic composition results in a table-driven implementation that, for each parallel call of a performance-aware component, looks up the expected best implementation variant, processor allocation and schedule given the current problem and processor group sizes. The dispatch tables are computed off-line at component deployment time by interleaved dynamic programming algorithm from time-prediction metacode provided by the component supplier.

1 Introduction

The multicore revolution in microprocessor architecture has just begun. Programmers will soon be faced with hundreds of hardware threads on a single-processor chip. Exploiting them efficiently is the only way to keep up with the performance potential that still follows the exponential development predicted by Moore's Law. This makes the introduction of parallel computing inevitable even in mainstream computing. Automatic parallelization is often not applicable due to the lack of static information or not efficient due to the principle overhead. Hence, explicit parallel programming is likely to become a dominating programming paradigm for the foreseeable future. This constitutes a challenge for programmers: in addition to the ever growing complexity of software, they now ought to master parallelism in an efficient and effective way, which creates a whole new problem dimension.

Current component technology fits well the domain of sequential and concurrent object-oriented programming. However, existing component models and composition techniques are poorly suited for the performance-aware composition of components for massively parallel computing. Classical component systems allow composition of functionality beyond language and platform boundaries but disregard the performance aspect. The HPC domain (and, as argued, very soon even mainstream computing) requires composition systems with explicitly parallel components. In particular, a performance-aware component model and composition technique are necessary to build complex yet efficient and scalable software for highly parallel target platforms.

In this paper, we propose a new approach for performance-aware composition of explicitly parallel components. We require that all parallel and sequential components subject to performance-aware composition adhere to a specific *performance interface* recognized by a special *performance-aware composition tool*. Performance-aware composition requires the component provider to supply metacode for each performance-aware method f. We focus on terminating methods f; the metacode is used to predict the execution time

of f. The metacode includes a `float`-valued function `time_`f depending on the number p of processors available and some selected parameters of f; `time_`f approximates the expected runtime of f used for local scheduling purposes. It is supplied by the component provider instead of computed by static analysis of f. In practice, it may interpolate entries in a precomputed table or use a closed form function but rather not simulated execution.

Functional and performance signatures are exposed by the component provider. Different substitutable component variants implementing the same functionality f inherit from the same interface. In general, different variants have different `time_`f functions. Dynamic composition chooses, for each call, the variant expected to be the fastest .

Parallel implementations may additionally exploit nested parallelism, marked explicitly by a parallel composition operator, putting together independent subtasks that could be executed in parallel or serially. Different schedules and processor allocations are precomputed, depending on static information like the characteristics of the hardware platform and the variants of subtasks available. They are then selected dynamically for different runtime configurations of problem sizes and processor group sizes.

2 Example: Parallel and Sequential Sorting Components

In the following example, we use pseudocode with object-oriented syntax and a shared-memory programming model similar to Fork[4] where statements and calls are executed in SPMD style, i.e., by a group of processors synchronized at certain program points. Note that our framework is by no means restricted to SPMD or object-oriented paradigms. In particular, it should work equally well with modular languages that provide the interface concept. Moreover, it would work equally well with a message passing programming model such as MPI, as we exemplified for parallel composition in earlier work[3].

We consider *sorting* as an example of functionality supported by several parallel and/or sequential components. We picked sorting since alternative sequential and parallel sorting algorithms such as quicksort, mergesort, or bitonic sort are well-known. Moreover, there are variants highly depending on the actual input (quicksort) and others that are less input-independent (mergesort). Hence, sorting comprises properties of a broad range of potential applications. All component variants conform to the same interface:

```
/* @performance_aware */ interface Sort {
  /* @performance_aware */ void sort( float *arr, int n ); }
```

The `performance_aware` qualifier marks the interface and its method `sort` for the composition tool. It expects a performance-aware implementation variant of `sort` and a time function `time_sort` with a subset of the parameters of `sort` in all implementation variants, e.g., a parallel quicksort:
The operator `compose_parallel` marks independent calls to the performance-aware method `sort`; these may be executed in parallel or serially, in any order. The composition tool will replace this construct with a dynamic dispatch selecting the approximated optimum schedule (serialization of calls, parallel execution by splitting the current group of processors into subgroups, or a combination of thereof) and implementation variant.

Within a `time_sort` function, the component designer specifies a closed formula, a table lookup, or a recurrence equation for the expected runtime of this variant. This meta-

```
/* @performance_aware */ component ParQS variantOf Sort {
  float find_pivot( float *arr, int n ) { ... }
  int partition( float *arr, int n, float pivot) {...}

  /* @performance_aware */ void sort( float *arr, int n ) {
      if (n==1) return;
      float pivot = find_pivot( arr, n );
      int n1 = partition( arr, n, pivot );
      /* @compose_parallel */
          /*@1*/  sort( arr, n1 );
          /*@2*/  sort( arr+n1, n-n1 );
      /* @end_compose_parallel */
  }

  /* @time_metadata */    // parsed and used by composition tool
    // aux. for time_sort, found empirically at deployment time
    const float T_test_1 = ...       // time for recursion end
    const float T_find_pivot = ...   // time for find_pivot
    const float[][] T_partition =... // table of partition times
    ...                              // benchmark metacode omitted
    float time_sort( int p, int n ) {
      if (n == 1) return T_test_1;
      float acctime=0.0;
      for (int n1=1; n1<n; n1++)
          acctime += TIMEpar( sort@1, n1, sort@2, n-n1, p );
      float exptime = acctime/(float)(n-1);
      return T_test_1 + T_find_pivot + T_partition[n][p] + exptime;
    }
  /* @end_time_metadata */
}
```

code is used to generate the dynamic dispatch tables. The operator `TIMEpar` computes the time for a parallel composition referring to independent calls, e.g., `sort@1` and `sort@2`, and the number `p` of processors available. It is the approximation of the makespan of the schedule found in optimization, cf. Section 3.

Below the `sort` and `time_sort` functions of a performance-aware sequential sorting component `SeqQS`, which simply wraps a (non-performance-aware) sequential quicksort library routine. For brevity, we omit the code of a parallel merge sort variant `ParMS`, which is also used in our example implementation, cf. Section 4.

```
  /* @performance_aware */ void sort( float *arr, int n ) {
     seq qsort( arr, n );  // done on 1 processor only
  }
  /* @time_metadata */    // used by composition tool
    const float[] T_qsort = ... // Table of seq. sorting times
    float time_sort( int P, int n ) { return T_qsort[n]; }
  /* @end_time_metadata */
```

3 Performance-Aware Composition

The optimization problem for composition is to determine (i) for each call to a performance-aware function, the (expected) best implementation variant, (ii) for each parallel composition operator in a performance-aware function, the number of processors to spend on the calls of each subtask, and a schedule for these subtasks.

The optimization problem may be reduced to the *independent malleable task scheduling* problem. A malleable task can be executed on $p = 1 \dots P$ processors. Its execution

time is described by a non-increasing function τ in the number of processors p actually used. A malleable task t_f corresponds to a call to a performance-aware interface function f, resp. its implementation variants. For each malleable task t_f, the τ_f-function is approximated using the `time_f` functions. Then, `compose_parallel` is replaced by selecting a schedule of independent malleable task.

Even without the choice between different variants, this scheduling problem is known to be NP-hard in the general case, but good approximations exist: for instance, k independent malleable tasks can be scheduled in time $O(P \cdot k^2)$ on P processors such that the completion time of the resulting schedule is at most $\sqrt{3}$ times the optimum[6]. Moreover, k *identical* malleable tasks can be scheduled optimally to P processors in time $O(\max(\log(k) \cdot t^{3P}, k \cdot (2t)^P)$ where t is the sequential execution time of a task[2].

Dispatch table generation The composition tool does not directly create the customized code. Instead, it generates (i) a variant dispatch table V_f for each interface function f and (ii) a schedule lookup table S for each parallel composition operator, listing the (expected) best processor allocation and the corresponding schedule. The table lists entries with the best decision for a range of problem sizes (ranging from 1 to some maximum tabulated problem size, suitably compacted) and a number of processors (ranging from 1 to the maximum number of processors available in the machine, suitably compacted). For our example above, $V_{\texttt{sort}}(n, p)$ contains a pointer to the expected best `sort` function for problem size n and processor group size p, – see Fig. 1 (right). $S(n_1, n_2, p)$ for the parallel composition operator in `ParQS` yields a processor allocation (p_1, p_2), where $p_1 + p_2 \leq p$, and a pointer to the expected best schedule variant, here only one of two variants: parallel or serial execution of the two independent calls to `sort`.

The tables V_f and S are computed by an *interleaved dynamic programming algorithm* as follows. Together with V_f and S, we will construct a table $\tau_f(n, p)$ containing the (expected) best execution times for p processors.

For a base problem size, e.g. $n = 1$, we assume the problems to be trivial and the functions not to contain recursive malleable tasks, i.e., no recursive calls to performance-aware functions. Hence, $\tau_f(1, p)$ can be directly retrieved from the corresponding $\texttt{time}_f$ functions and $V_f(1, p)$ selected accordingly as the variant with minimum $\texttt{time}_f$.

For $p = 1$, parallel composition leads to a sequential schedule, i.e., the entries $S(..., 1)$ point to sequential schedules where each task uses 1 processor. Hence, no performance-aware function contains alternative schedules for $p = 1$ and `TIMEpar` reduces to a simple addition of execution times of the subtasks. Accordingly, the $\tau_f(n, 1)$ and $V_f(n, 1)$ for $n = 1, 2, ...$ can be derived iteratively from the $\texttt{time}_f$ functions: $\tau_f(n, 1)$ is set to the minimum $\texttt{time}_f$ of all variants of f and $V_f(n, 1)$ is set to the variant with minimum $\texttt{time}_f$. Usually, the sequential variants (not containing a parallel composition at all) outperform the serialized parallel variants.

Then, we calculate the remaining table entries stepwise for $p = 2, 3, ...$ For each p, we consider successive $n = 1, 2, ...$ For each such n, we determine $\tau_f(n, p)$, $V_f(n, p)$, and the schedules of sub-problems $S(n_1, n_2, \ldots, n_k, p)$. Hence, for $n > 1, p > 1$, we have already computed $\tau_f(n', p')$, $V_f(n', p')$ and $S(n_1, n_2, \ldots, n_k, p')$ with $n', n_1, n_2, \ldots, n_k < n$ and $p' \leq p$.

First, we calculate the schedules $S(n_1, n_2, \ldots, n_k, p)$ for the `compose_parallel` constructs: Since τ_f is defined for each call contained, we simply apply an approximation to the independent malleable task problem leading to a schedule and processor allocations

$(p_1, p_2, \ldots, p_k)$, where all $p_i \leq p$. In our example, we do not even need to approximate the optimum since there is only the parallel schedule with finitely many parallel allocations $(p_1, p - p_1)$ and the sequential one.

Second, we compute $\tau_f(n,p)$ and $V_f(n,p)$: We replace the `TIMEpar` construct with the makespan of the schedule derived and evaluate the $\texttt{time}_f$ functions for each variant. Again, $\tau_f(n,p)$ is set to the minimum $\texttt{time}_f$ of all variants of f and $V_f(n,p)$ is set to the variant with minimum $\texttt{time}_f$.

Composition Auxiliary performance functions and tables as needed for the $\texttt{time}_f$ functions are determined at component deployment time. Accordingly, a component variant provider needs to provide benchmark metacode executed before optimization.

All performance-aware components for the same interface should be deployed together to get meaningful table entries. Encapsulation is still preserved as third-party component providers need not know about other performance-aware components that co-exist.

After optimization, the composition tool patches each call to f in all component implementations with special dispatch code that looks up the variant to call by inspecting the V_f table at runtime, and generates dispatch code at each parallel composition operator looking up its S table at runtime with the current subproblem and group size to adopt the (expected) best schedule.

The example code for `ParQS` after composition is sketched below:

```
component ParQS {
  float find_pivot( float *arr, int n ) { ... }
  int partition( float *arr, int n, float pivot) {...}

  extern const int[][] V_sort;      // The V table
  const int[][][][] S_sort = ...    // The S table
  const void (*Sched_sort[2])(float *, int) = { s1_sort, s2_sort };

  void sort( float *arr, int n ) {
      if (n==1) return;
      float pivot = find_pivot( arr, n );
      int n1 = partition( arr, n, pivot );
      //Schedule dispatch - look up function pointer in S and call:
      int p = groupsize(); // number of executing processors
      Sched_sort[ S[n1][n-n1][p][0] ]
         ( arr, n1, n-n1, S[n1][n-n1][p][1], S[n1][n-n1][p][2] );
  }
  //serialized schedule:
  void s1_sort( float *arr, int n1, int n2, int p1, int p2 ) {
      V_sort[n1][p1] ( arr, n1, p1 );
      V_sort[n-n1][p2] ( arr+n1, n2, p2 );
  }
  //parallel schedule:
  void s2_sort( float *arr, int n1, int n2, int p1, int p2 ) {
      split_group(p1,p2) { V_sort[n1][p1](arr, n1); }
                         { V_sort[n2][p2](arr+n1, n2);}
  }
}
```

The table entry `S_sort[n1][n2][p][0]` contains the precomputed schedule variant, the entry `S_sort[n1][n2][p][i]` the processor allocation for the `i`-th call. `groupsize()` returns the number of processors in the executing group. Any other call to `sort(a,n)` would be patched to `V_sort[n][groupsize()](a,n)`. Technically, the composition tool could be based on COMPOST[1] or a similar tool for static meta-

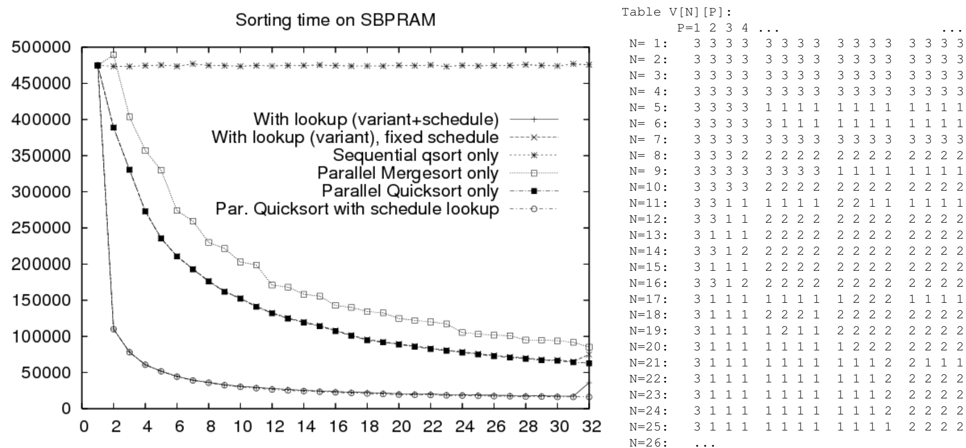

Table V[N][P]:

	P=1	2	3	4	...											...
N= 1:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N= 2:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N= 3:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N= 4:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N= 5:	3	3	3	3	1	1	1	1	1	1	1	1	1	1	1	1
N= 6:	3	3	3	3	3	1	1	1	1	1	1	1	1	1	1	1
N= 7:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N= 8:	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2
N= 9:	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1	1
N=10:	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2
N=11:	3	3	1	1	1	1	1	1	2	2	1	1	1	1	1	1
N=12:	3	3	1	1	2	2	2	2	2	2	2	2	2	2	2	2
N=13:	3	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
N=14:	3	3	1	2	2	2	2	2	2	2	2	2	2	2	2	2
N=15:	3	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
N=16:	3	3	1	2	2	2	2	2	2	2	2	2	2	2	2	2
N=17:	3	1	1	1	1	1	1	1	1	2	2	2	1	1	1	1
N=18:	3	1	1	1	2	2	2	1	2	2	2	2	2	2	2	2
N=19:	3	1	1	1	1	2	1	1	2	2	2	2	2	2	2	2
N=20:	3	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
N=21:	3	1	1	1	1	1	1	1	1	1	1	2	2	1	1	1
N=22:	3	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
N=23:	3	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
N=24:	3	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
N=25:	3	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
N=26:	...															

Figure 1. Execution times (in SBPRAM clock cycles) of the composed sorting program, applied to $N = 500$ numbers on up to $P = 32$ SBPRAM processors, compared to the times needed when using each component exclusively. The dispatch tables were precomputed by dynamic programming for $N = 1...64$ and $P = 1...32$. — Right-hand side: The top-left-most entries (1..25×1..16) in the variant dispatch table V indicating the expected best variant, where 1 denotes a call to ParQS, 2 to ParMS, and 3 to SeqQS.

programming that enables fully programmable program restructuring transformations.

4 Implementation and First Results

A proof-of-concept implementation uses the C-based parallel programming language Fork[4] for three sorting components, parallel recursive quicksort, parallel recursive mergesort, and sequential quicksort. The time parameters for the functions `find_pivot`, `partition`, `qsort` etc. used in `time_sort` were obtained by measuring times for example instances. Times depending on input data were averaged over several instances. To record the best schedules for parallel composition in the schedule table, a brute-force enumeration approach was chosen, as the best constellation of subgroup sizes can be determined in linear time for a parallel divide degree of 2, as in `ParQS` and `ParMS`.

As a fully automatic composition tool interpreting metacode syntax is not yet available, we simulated the effect of composition by injecting the the dispatch tables and lookups by hand into the appropriate places in the source code. For evaluation purposes, the resulting Fork source code can be configured to either use the schedule and variants as given in the computed dispatch tables or the original component implementations without modification. The code is compiled to the SBPRAM[4] and executed on its cycle-accurate simulator, run on a SUN Solaris server.

Figure 1 shows average times for sorting 500 numbers on up to 32 PRAM processors (left) and a section of the variant dispatch table V (right). We can observe that all parallel variants outperform sequential quicksort. Among the former, adaptive parallel quicksort and our performance-aware composition perform best. That these two variants perform almost equivalent comes at no surprise when looking the V table entries: except for small problem sizes, quicksort is selected.

The performance improvements of the composed function (up to a factor of 10 compared to sequential sorting and up to a factor of 4 and 5 compared to the parallel quicksort and mergesort only, resp.) are due to schedule lookup. The gain increases with N because the general case of two recursive subtasks gets more common.

5 Related Work

Various static scheduling frameworks for malleable parallel tasks and task graphs of modular SPMD computations with parallel composition have been considered in the literature[7–9]. Most of them require a formal, machine-independent specification of the algorithm that allows prediction of execution time by abstract interpretation. In our work, we separated the actual implementation from the model of its execution time. Scheduling methods for distributed memory systems also need to optimize communication for data redistribution at module boundaries. This can be added to our framework as well. To the best of our knowledge, none of them considers the automatic composition of different algorithm variants at deployment time and their automatic selection at runtime.

As we do not consider heterogeneous or distributed systems here, additional interoperability support by parallel CORBA-like systems is not required. However, such an extension would be orthogonal to our approach.

Our earlier work explores the parallel composition operator for dynamic local load balancing in irregular parallel divide-and-conquer SPMD computations[3]. We balanced the trade-off between group splitting for parallel execution of subtasks and serialization, computing—off-line by dynamic programming—tables of the expected values of task size ratios, indexed by n and p, where scheduling should switch between group splitting and serialization. Our schedule lookup table above can be seen as a generalization of this.

6 Conclusions and Future Work

The paper proposes a composition framework for SPMD parallel components. Components are specified independently of the specific runtime environment. They are equipped with metacode allowing to derive their performance in a particular runtime (hardware) environment at deployment time. Based on this information, a composition tool automatically approximates optimal partial schedules for the different component variants and processor and problem sizes and injects dynamic composition code. Whenever the component is called at runtime, the implementation variant actually executed is selected dynamically, based on the actual problem size and the number of processors available for this component. Experiments with two parallel and one sequential sorting component prototypically demonstrate the speed-up compared to statically composed parallel solutions.

Static agglomeration of dynamic composition units is an optimization of our approach. We could consider the trade-off between the overhead of dynamic composition vs. the (expected) performance improvement due to choosing the (expected) fastest variant and schedule. We could consider units for dynamic composition that have a larger granularity than individual performance-aware function calls. A possible approach could be to virtually in-line composition operator "expressions" (which may span across function calls, i.e., define contiguous subtrees of the call graph) that will be treated as atomic units for dynamic composition. The composition tool would compose these units statically including

a static composition of the `time` functions. This will usually somewhat decrease accuracy of predictions and miss some better choices of variants within these units but also saves some dynamic composition overhead.

Table compression techniques need to be investigated. For instance, regions in the V or S tables with equal behaviour could be approximated by polyhedra bounded by linear inequalities that could result in branching code instead of the table entry interpolations for dynamic dispatch and scheduling. Compression techniques for dispatch tables of object-oriented polymorphic calls could be investigated as well.

Adaptation of time data parameters In the sorting example, we used randomly generated problem instances to compute parameter tables with average execution times for `qsort`, `partition` etc. used in optimization. In certain application domains or deployment environments, other distributions of input data could be known and exploited. Moreover, expected execution times could be adjusted dynamically with new runtime data as components are executed, and in certain time intervals, a re-optimization may take place such that the dispatch tables adapt to typical workloads automatically.

Domains of application In scientific computing as well as non-numerical applications, there are many possible application scenarios for our framework. For instance, there is a great variation in parallel implementations of solvers for ODE systems that have equal numerical properties but different time behaviour[5].

Acknowledgements Research funded by Ceniit 01.06 at Linköpings universitet, Vetenskapsrådet, SSF RISE, Vinnova SafeModSim and AdaptiveGRID, and the CUGS graduate school.

References

1. U. Aßmann, *Invasive Software Composition*, (Springer, 2003).
2. T. Decker, T. Lücking and B. Monien, *A 5/4-approximation algorithm for scheduling identical malleable tasks*, Theoretical Computer Science, **361**, 226–240, (2006).
3. M. Eriksson, Ch. Kessler and M. Chalabine, *Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments*, in: Proc. 8th Workshop on Parallel Systems and Algorithms (PASA'06), GI Lecture Notes in Informatics (LNI), vol. **P-81**, pp. 313–322, (2006).
4. J. Keller, Ch. Kessler and J. Träff, *Practical PRAM Programming*, (Wiley Interscience, 2001).
5. M. Korch and Th. Rauber, *Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining*, J. Par. and Distr. Computing, **66**, 444–468, (2006).
6. G. Mounie, C. Rapine and D. Trystram, *Efficient approximation algorithms for scheduling malleable tasks*, in: Proc. 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99), ACM Press, pp. 23–32, (1999).
7. Th. Rauber and G. Rünger, *Compiler Support for Task Scheduling in Hierarchical Execution Models*, J. Systems Architecture, **45**, 483–503, (1998).
8. L. Zhao, S. Jarvis, D. Spooner and G. Nudd, *Predictive Performance Modeling of Parallel Component Composition*, in: Proc. 19th IEEE Int. Parallel and Distr. Processing Symposium (IPDPS-05), (IEEE Press, 2005).
9. W. Zimmermann and W. Löwe, *Foundations for the integration of scheduling techniques into compilers for parallel languages*, Int. J. Comp. Sci. Eng., **1**, 3/4, (2005).