Applicability of Load Balancing Strategies to Data-Parallel Embedded Runge-Kutta Integrators

Matthias Korch and Thomas Rauber

University of Bayreuth, Department of Computer Science
{matthias.korch, rauber}@uni-bayreuth.de

Abstract. Embedded Runge-Kutta methods are among the most popular methods for the solution of non-stiff initial value problems of ordinary differential equations (ODEs). We investigate the use of load balancing strategies in a data-parallel implementation of embedded Runge-Kutta integrators. Since the parallelism contained in the function evaluation of the ODE system is typically very fine-grained, our aim is to find out whether the employment of load balancing strategies can be profitable in spite of the additional overhead they involve.

1 Introduction

In this paper, we consider the parallel solution of initial value problems (IVPs) of ordinary differential equations (ODEs) defined by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \qquad \mathbf{y} : \mathbb{R} \to \mathbb{R}^n, \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n. \tag{1}$$

The numerical solution of large IVPs is a computationally intensive task. Therefore, efforts have been taken to find efficient parallel solution methods, e.g., extrapolation methods [1], waveform relaxation techniques [2], and iterated Runge-Kutta methods [3]. Most of these approaches develop new numerical algorithms with a larger potential for parallelism, but with different numerical properties.

Non-stiff ODE systems can be solved efficiently by embedded Runge-Kutta (ERK) methods with stepsize control. Popular methods are, for example, DOPRI5(4) and DOPRI8(7) [4]. An ERK method with s stages which uses the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ to compute the two new approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ from the two previous approximations η_κ and $\hat{\eta}_\kappa$ is represented by the computation scheme

$$\begin{gathered} \mathbf{w}_l = \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{w}_i), \quad l = 1, \ldots, s, \\ \eta_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{w}_l), \quad \hat{\eta}_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{w}_l). \end{gathered} \tag{2}$$

The coefficients a_{li}, c_i, b_l, and $\hat{b}_l$ are determined by the particular ERK method used.

Because, in general, all coefficients a_{ij} may be non-zero and an evaluation of the right hand side function $\mathbf{f}(t, \mathbf{w})$ may access all components of the argument vector $\mathbf{w}$, the stages $l = 1, \ldots, s$ have to be computed sequentially. However, ERK methods possess a large potential for data-parallelism across the ODE system, since the function

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 720–729, 2006.

evaluations of individual ODE components can be performed in parallel. Experiences from earlier experiments (e.g., [5,6], cf. [2]) suggest that this type of parallelism can be exploited efficiently only if the ODE system is sufficiently large and the communication network of the parallel computer system is fast in relation to the speed of the processors or the function evaluations of the ODE components are computationally intensive. In general, the obtainable performance strongly depends on the characteristics of the IVP (cf. Section 5). But if these conditions are fulfilled, general ERK solvers can work efficiently on small or medium-sized shared-memory multiprocessors (SMMs). However, on larger SMMs and on most modern distributed-memory multiprocessors (DMMs) the speedups obtainable with current implementations are not yet satisfactory [6].

Therefore, it is desirable to find new implementations that can deliver a higher efficiency. Two possible approaches take advantage of special properties of either the ERK method [7] or the ODE system [5,6]. In this paper, we follow a different approach which requires no assumptions about particular properties of the method or the ODE system. We investigate if an improvement in performance can be achieved by the application of dynamic load balancing strategies. We show that if the load balance can be achieved with only little overhead, a higher performance can be obtained if the right hand side function of the problem is irregular and also in other situations where a load imbalance limits scalability, while the performance for regular problems is still competitive with solvers with a static work distribution.

2 Motivation and Computational Structure

Knowing from previous experiments that the communication costs of general ERK solvers on DMMs are to high for most problems to achieve satisfactory speedups, we concentrate our initial investigations of load balancing strategies on SMMs, because on such machines load balancing strategies with little overhead can be realized.

The computational kernel of a data-parallel implementation of a general ERK solver with a static blockwise data distribution can be realized as shown in Fig. 1. Since the evaluation of the right hand side function $\mathbf{f}$ cannot start before the parallel computation of the corresponding argument vector has been completed, a barrier operation must be executed before each stage. Since no further synchronization operations are used, the scalability is mainly determined by the efficiency of the barriers, the waiting times due to memory operations and the waiting times of the processors at the barriers.

In practice, the processors may not arrive at the barriers simultaneously for several reasons: (1) The function $\mathbf{f}$ can be irregular, i.e., a different number of instructions is required to evaluate the individual components. Examples are shown in Section 3. (2) A parallel computer can be heterogeneous, i.e., the processors work at different speeds. (3) The operating system scheduler may temporarily suspend threads of the ERK solver to execute other processes. (4) On systems with non-uniform memory access times (NUMA), the latency of memory operations can vary between one and several thousands of cycles. (5) On systems with simultaneous multithreading (SMT) support, the threads of the ERK solver compete for the functional units of the processors.

Considering these facts, asynchronous techniques that adaptively assign work to the participating processors might be able to improve the performance. Therefore, based on

```
 1: me := my_thread_id;

 2: barrier();
 3: for (j := first_component[me]; j ≤ last_component[me]; j++)
 4:     v := hf_j(t + c_1 h, η);
 5:     for (i := 2; i ≤ s; i++) w_i[j] := η[j] + a_{i1} v;
 6:     η_{κ+1}[j] := b_1 v; η̂_{κ+1}[j] := b̂_1 v;

 7: for (l := 2; l ≤ s; l++)
 8:     barrier();
 9:     for (j := first_component[me]; j ≤ last_component[me]; j++)
10:         v := hf_j(t + c_l h, w_l);
11:         for (i := l + 1; i ≤ s; i++) w_i[j] += a_{il} v;
12:         η_{κ+1}[j] += b_l v; η̂_{κ+1}[j] += b̂_l v;
```

Fig. 1. Computational kernel of a data-parallel ERK implementation for shared address space using a static blockwise data distribution

our experience with load balancing of task-based irregular applications [8,9] we have implemented different strategies that realize a dynamic work distribution and apply them to several test problems. Our aim is to investigate if and under which conditions a performance improvement can be achieved on modern SMMs, and which performance bottlenecks still remain to be resolved.

3 Test Problems

We consider four test problems which exhibit different characteristics and are therefore suitable for the investigation of different aspects of our load balancing strategies. Figure 2 shows the number of instructions and the number of cycles required to evaluate the individual components of these problems on a Pentium 4 processor.

- **EMEP** [10] is the chemistry part of the EMEP-MSC-W ozone chemistry model. The dimension of this problem is 66. This problem exhibits the most irregular structure of all problems in our testset because the equations that model the concentrations of the individual species have a widely varying complexity.
- **MEDAKZO** [10]. The medical Akzo Nobel problem has been derived from two 1D partial differential equations (PDEs) which describe the penetration of antibodies into a tissue that has been infected by a tumor. The dimension of this system $n = 2N$ depends on the discretization parameter N. The number of instructions required to evaluate the ODE components differs between odd and even components. But if a blockwise data distribution is used, the load is nearly evenly balanced.
- **STARS** [11,12] describes a 3D n-body problem. The original second order ODE system has been transformed into a first order system by substitution of the first derivative. The system dimension is $6N$, where N is the number of stars. We consider two orderings of the components: STARS-CON uses a consecutive ordering of the first and second derivative and leads to a very uneven load balance. STARS-MIX interleaves the two derivatives and thus balances the load more evenly.

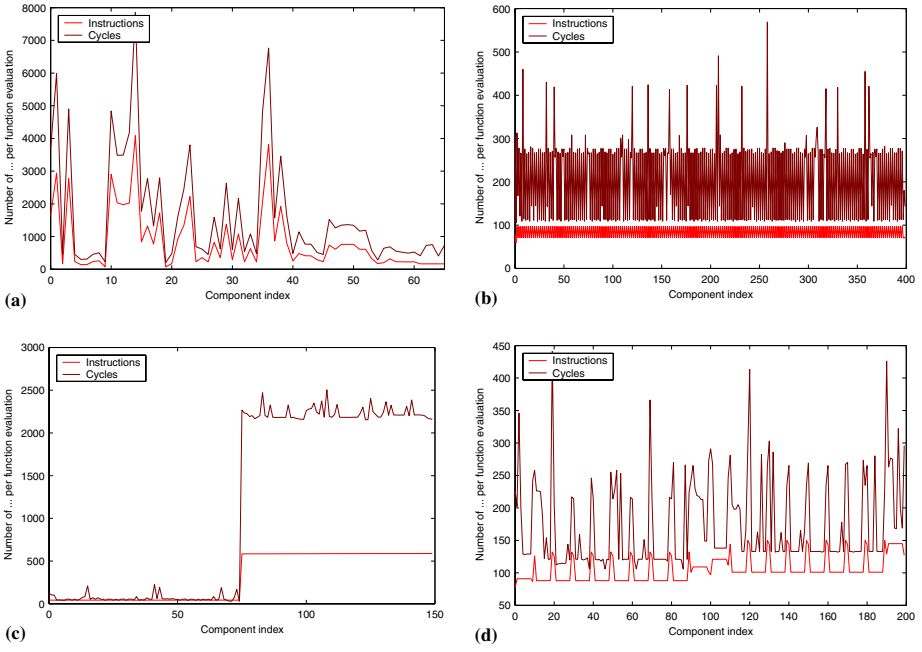

Fig. 2. Number of instructions and number of cycles required to evaluate the individual components of some test problems on a Pentium 4 processor. **(a)** EMEP, **(b)** MEDAKZO, $N = 200$, **(c)** STARS-CON, $N = 25$, **(d)** BRUSS2D-ROW, $N = 10$.

– **BRUSS2D** [13] results from a 2D PDE system that describes the chemical reaction of two substances. We consider two orderings: a row-oriented ordering, BRUSS2D-ROW, where the concentrations of the two substances are stored consecutively, and a mixed row-oriented ordering, BRUSS2D-MIX, where the components of the two substances are interleaved. The evaluation costs of the components of the two substances are slightly different. Elements at the boundary of the discretization grid require a special treatment. The balance of the decision tree used to identify boundary elements influences the regularity of the ODE system. The decision tree realized for BRUSS2D-MIX is more evenly balanced than that of BRUSS2D-ROW.

4 Load Balancing Strategies

The realization of a load balancing strategy for an ERK solver leads to the problem of scheduling the iterations of the irregular loops in lines 3 and 9 of Fig. 1 dynamically. The problem of loop scheduling has been considered previously by several authors, e.g., [14,15]. We have implemented three different load balancing strategies that pay regard to the special context of the loops within the ERK solver. All strategies start with the same blockwise work distribution as the static implementation. But after a processor has finished its own range of components, it ‘steals’ work from other processors and

```
 1: me := my_thread_id;
 2: for (l := 1; l ≤ s; l++)
 3:     next_work_unit_l[me] := first_work_unit[me];

 4: for (l := 1; l ≤ s; l++)
 5:     barrier();
 6:     loop
 7:         work_unit := FETCH_AND_INC(
 8:             next_work_unit_l[me]);
 9:         if (work_unit > last_work_unit[me])
10:             me := NEXT_THREAD_ID(me);
11:             if (me = my_thread_id) break;
12:         else
13:             PROCESS_WORK_UNIT(l,
    work_unit);
```

Fig. 3. Load balancing strategy 'Simple'

Table 1. Overview of the load balancing implementations

Name	Strategy	Granularity	Synchronization
SCIL	simple/increment	components	lock
SPIL	simple/increment	cache lines	lock
SCIA	simple/increment	components	atomic operations
SPIA	simple/increment	cache lines	atomic operations
SCRA	simple/random	components	atomic operations
SPRA	simple/random	cache lines	atomic operations
TC	task queue	components	lock
TP	task queue	cache lines	lock
IC	interval queue	components	lock
IP	interval queue	cache lines	lock

thus 'helps' these processors that would otherwise arrive late at the next barrier. The load balancing strategies differ by the data structures used to represent tasks, i.e., work units and by the policies used to steal work. All strategies have been implemented in C with POSIX Threads in two versions that support two different task granularities: single components and a group of components that fit into one cache line. Table 1 shows an overview of all load balancing implementations discussed in this article.

Strategy 'Simple'. As a simple but effective load balancing strategy with small overhead we have realized a strategy that was also used in the *volrend* application [16] included in the SPLASH-2 benchmark suite. Every thread provides a counter that points to the next work unit to be processed in the range initially assigned to the thread. During execution, each thread fetches and increments the counter corresponding to its thread ID. As long as the counter points to a work unit that lies within the range assigned to its thread ID, the thread processes the corresponding work unit and then fetches and increments the counter again. If the value of the counter leaves the range assigned to the thread ID, the thread changes its thread ID and continues with the corresponding counter. The pseudocode of this algorithm is displayed in Fig. 3.

We have implemented two strategies for changing the thread ID: 'Increment' computes the new ID as in [16] by $((\text{old\_id} + 1) \bmod \#\text{threads})$. 'Random' reduces the probability that many threads work on the same counter simultaneously by changing the thread ID according to an initially generated random permutation.

Since the counters can be accessed by several threads simultaneously, we must ensure that the threads are not preempted when they read and increment the counters. We achieve this by either using locks or atomic Fetch & Inc. The lock based implementations use mutex variables of type `pthread_spinlock_t` if available, e.g., on Linux-based systems, or `pthread_mutex_t` on other systems. The implementations based on atomic Fetch & Inc currently support the following platforms:

- IA64: Fetch & Add is available as a machine instruction.
- x86: We emulate Fetch & Inc by a loop using Compare & Swap.
- AIX: The operating system kernel provides the function `fetch_and_add()`.

Strategy 'Task Queue'. This is a more sophisticated strategy than 'Simple' that reduces contention but requires a higher sequential overhead. It realizes one task queue per thread with FIFO (first in, first out) access order. The tasks are represented by integer values specifying the index of the associated work unit. Hence, the FIFO queues can be implemented by fixed size arrays of integers with head and tail pointers. Locks (`pthread_mutex_t` or `pthread_spinlock_t` if available) are used to avoid race conditions when the queues are accessed by several processors simultaneously.

The load balancing algorithm based on the 'Task Queue' strategy is shown in Fig. 4. At the beginning of each stage the queues are initialized according to the same block-wise work distribution as used in the static implementation. But the size of the queues is not revealed to the other threads before the barrier has been executed, so that no other thread that still works on the preceding stage will steal work from a queue during this initialization phase. After the initialization, the threads fetch tasks from the head of their local queues until their local queues get empty. When a thread finds no more tasks in its own queue, it tries to 'steal' work from another thread, i.e, it tries to move tasks from another thread's queue into its own queue. The stealing heuristics tries to steal half of the average queue size tasks from the tail of the queue with the largest size. The stolen tasks are appended at the tail of the target queue. But before the tasks are removed from the source queue, the size of the source queue is decreased by the number of tasks to be stolen, so that it appears less attractive to other threads searching for work. The new size of the target queue is hidden to the other threads, thus pretending it were still empty until all tasks have been transferred. Therefore, no thread will try to steal work from the target queue during the task transfer.

Strategy 'Interval queue'. Analyzing the behavior of the 'Task Queue' strategy we observe that the queues always store consecutive intervals of work units. A different data structure, called interval queue, can provide similar operations as the task queue but stores only the lowest and the highest index of the range of work units contained in the queue. It therefore leads to a significantly lower overhead. To fetch a work unit, the local thread only needs to increment the start index of the interval by 1; m tasks can be stolen at once in time $O(1)$ by decrementing the end index of the interval by m. Hence, the load balancing algorithm based on the interval queue can be realized similarly as in Fig. 4 by replacing lines 4–6 by

```
REWIND(my_queue, first_work_unit[me], last_work_unit[me]);
```

and lines 21–28 by

```
work_unit := STEAL(queue[index], my_queue, work_units_to_steal);
UNLOCK(queue[index]);
```

where REWIND(Q, A, B) initializes the interval stored in queue Q to $[A, B]$ and STEAL(Q_S, Q_T, m) decreases the end of the interval of the source queue Q_S by m, initializes the target queue Q_T with the stolen interval of size m and returns the first element of this interval.

```
1:  me := my_thread_id;
2:  my_queue := queue[me];

3:  for (l := 1; l ≤ s; l++)
4:      REWIND(my_queue);
5:      for (j := first_work_unit[me]; j ≤ last_work_unit[me]; j++)
6:          HIDDEN_APPEND(my_queue, j);
7:      barrier();
8:      REVEAL_SIZE(my_queue);
9:      loop
10:         LOCK(my_queue);
11:         if (EMPTY(my_queue))
12:             UNLOCK(my_queue);
13:             loop
14:                 sum := Σ_{k=1}^{#threads} SIZE(queue[k]);
15:                 index := argmax_{1≤k≤#threads} SIZE(queue[k]);
16:                 if (sum = 0) goto Stage_Complete;
17:                 if (TRYLOCK(queue[index]))
18:                     if (SIZE(queue[index]) > 0) break;
19:                     UNLOCK(queue[index]);
20:             work_units_to_steal := STEAL_HEURISTICS(SIZE(queue[index]), sum);
21:             PREPARE_STEALING(queue[index], work_units_to_steal);
22:             work_unit := STEAL(queue[index]);
23:             REWIND(my_queue);
24:             for (k := 1; k ≤ work_units_to_steal; k++)
25:                 HIDDEN_APPEND(my_queue, STEAL(queue[index]));
26:             FINISH_STEALING(queue[index], work_units_to_steal);
27:             UNLOCK(queue[index]);
28:             REVEAL_SIZE(my_queue);
29:         else
30:             work_unit := FETCH(my_queue);
31:             UNLOCK(my_queue);
32:         PROCESS_WORK_UNIT(l, work_unit);
33:     label Stage_Complete;
```

Fig. 4. Load balancing strategy 'Task Queue'

5 Runtime Experiments

Runtime experiments with the implemented load balancing strategies have been performed on three symmetric multiprocessors (SMPs): a 4-way 2.0 GHz Opteron 270 SMP, a 4-way 1.5 GHz Itanium 2 SMP, and an IBM p690 with 32 POWER4+ cores at 1.7 GHz. As a basis for the assessment of the implementations we use the average execution time per step measured by executing a limited number of time steps and dividing the resulting execution time by the number of steps executed. As a reference for the speedup calculation and the evaluation of the overhead we use a sequential implementation similar to Fig. 1, which contains no synchronization operations. All experiments presented in the following have been performed using the ERK method DOPRI5(4).

Sequential Overhead. First, we investigate the sequential overhead of the parallel implementations, that is the percentage of time they run slower than a sequential implementation when executed on one processor. As an example, Table 2 shows the sequential overhead measured for BRUSS2D-ROW with $N = 1000$. For this problem the highest overheads have been observed.

Table 2. Overhead of the parallel implementations for BRUSS2D-ROW with $N = 1000$ in %

Target system	static	IC	IP	SCIA	SCIL	SCRA	SPIA	SPIL	SPRA	TC	TP
Opteron 270 SMP	0.4	37.7	7.7	41.5	40.7	41.1	9.0	9.3	9.1	53.1	9.8
Itanium 2 SMP	4.7	29.6	6.2	10.9	29.8	10.9	5.0	6.1	4.9	35.0	6.8
IBM p690	5.1	274.4	37.4	131.6	266.8	124.7	10.8	21.5	6.4	299.2	42.2

Comparing the different target systems, the highest overheads have been measured on the IBM p690. On this system, `pthread_spinlock_t` is not available and we have to use `pthread_mutex_t` instead. Further, we used an AIX kernel function to realize atomic Fetch & Inc instead of inline assembler instructions as on the other two systems. Comparing the Itanium 2 and the Opteron SMP, we observe lower overheads on the Itanium 2 system. Hence, it appears that the spinlocks require less instructions on the Itanium 2 than on the Opteron. Also, the implementations that use atomic operations to realize Fetch & Inc run more efficiently on the Itanium 2 since on this machine only one machine instruction needs to be executed while on the Opteron more instructions are required to emulate Fetch & Inc by Compare & Swap. Only the overhead of our reference implementation based on a static work distribution is higher on the Itanium 2 than on the Opteron. This is due in part to cache effects caused by interferences between memory accesses to data and instructions.

In general, on all machines the overhead of the load balancing implementations is higher than that of the static implementation. The highest overhead was observed for the implementations that use single components as work units. The lowest overhead of the load balancing implementations is obtained by the implementations based on the 'Simple' strategy which use atomic operations to realize Fetch & Inc.

Scalability on the Itanium 2 SMP. In this section, we give a detailed discussion on the results measured on the Itanium 2 SMP. Because on this system the overhead of the load balancing implementations is lower than on the other two systems, we can observe relatively high improvements over the static work distribution. The speedup diagrams for the problems discussed in the following are shown in Fig. 5.

The most irregular test problem is EMEP. Due to its low dimension, only small speedups can be obtained. The best speedup of the static work distribution is 1.14 obtained on two processors. The implementations of 'Simple' that use cache lines as work units obtain a slightly better speedup of up to 1.17 on three processors. Using MEDAKZO with $N = 2400$ we obtain significantly better speedups. Except for TC and SCIL, all implementations obtain their maximum speedup on four processors. Using the static work distribution, a speedup of 2.74 is possible. But all load balancing implementations that use cache lines as work units obtain higher speedups. The best speedups between 3.02 and 3.06 are achieved by IP, SPRA, and SPIA. For STARS-CON and STARS-MIX we use $N = 1000$ stars. The load balancing implementations obtain nearly perfect speedups between 3.95 and 3.99 for STARS-CON and between 4.00 and 4.01 for STARS-MIX. With the static work distribution, only a speedup of 2.00 can be obtained for STARS-CON due to the severe load imbalance, but for STARS-MIX a speedup of 4.01 has been measured. For BRUSS2D with $N = 1000$ the speedups of the two orderings are similar. The static implementation obtains speedups of 3.32 and 3.30,

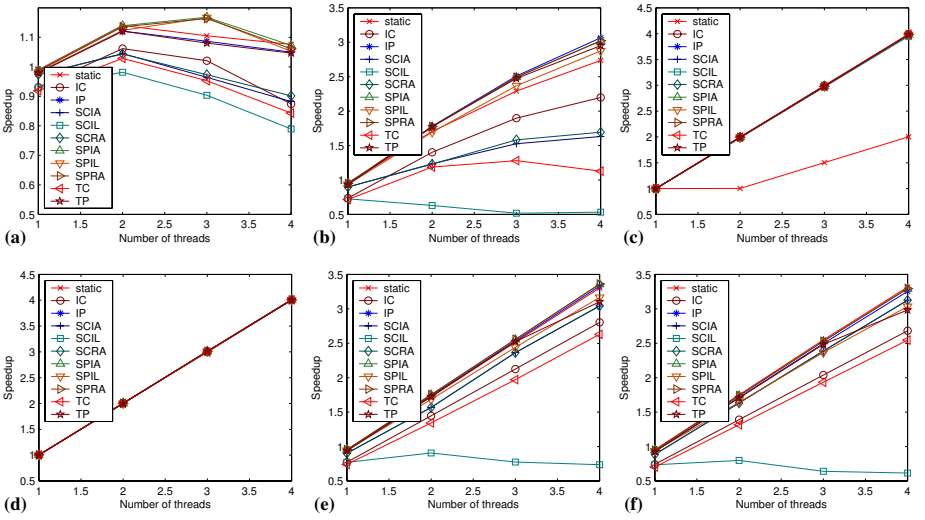

Fig. 5. Speedups measured on the Itanium 2 SMP. **(a)** EMEP, **(b)** MEDAKZO, $N = 2400$, **(c)** STARS-CON, $N = 1000$, **(d)** STARS-MIX, $N = 1000$, **(e)** BRUSS2D-ROW, $N = 1000$, **(f)** BRUSS2D-MIX, $N = 1000$.

respectively, for the two orderings on four processors. Since BRUSS2D-MIX is nearly evenly balanced, the best load balancing implementations, SPRA and SPIA, only obtain a slightly worse speedup of 3.29 for this ordering. But for BRUSS2D-ROW these two implementations obtain a better speedup than the static implementation of 3.36.

Scalability on Other Systems. The speedups measured on the Opteron 270 SMP and the IBM p690 are summarized in Table 3. On the Opteron system the load balancing implementations are similarly successful as on the Itanium 2 SMP. Thus, except for MEDAKZO, for every problem, at least one load balancing implementation obtains a higher speedup than the static work distribution. But on the IBM p690, due to a higher overhead, the load balancing implementations cannot obtain a higher performance than the static work distribution for most problems. Only for STARS-CON, which is characterized by a severe load imbalance, a significantly better speedup can be achieved.

Table 3. Summary of the speedups measured on the Opteron 270 SMP and on the IBM p690

		Opteron 270 SMP			IBM p690		
		Static	Load balancing		Static	Load balancing	
Problem	Parameter	Speedup	Speedup	Best implementation	Speedup	Speedup	Best implementation
EMEP		1.37	1.50	SPIA	1.35	1.35	TP
MEDAKZO	$N = 2400$	3.17	3.07	SPIA	3.23	2.88	SPRA
STARS-CON	$N = 1000$	1.97	3.98	IC, IP, TC, TP	15.65	30.13	SCRA
STARS-MIX	$N = 1000$	3.94	3.98	all except SCIL	30.60	30.24	SCIA
BRUSS2D-ROW	$N = 1000$	2.10	2.11	IC, IP	27.19	25.33	SPRA
BRUSS2D-MIX	$N = 1000$	2.26	2.28	IP	25.03	23.54	SPRA

6 Conclusions

Our results show that load balancing strategies can successfully be applied to data-parallel ERK solvers even though they require a larger sequential overhead than a static work distribution. They are particularly successful for ODE systems which lead to a severe load imbalance, but if special machine instructions are exploited to reduce the overhead, an improvement can be obtained even for some problems with a well-balanced right hand side function. However, our current load balancing implementations leave room for improvements, and a further investigation of load balancing strategies might lead to new insights.

References

1. Ehrig, R., Nowak, U., Deuflhard, P.: Massively parallel linearly-implicit extrapolation algorithms as a powerful tool in process simulation. In D'Hollander, E.H., et al., eds.: Parallel Computing: Fundamentals, Applications and New Directions. Elsevier (1998) 517–524
2. Burrage, K.: Parallel and Sequential Methods for Ordinary Differential Equations. Oxford Science Publications (1995)
3. van der Houwen, P.J., Sommeijer, B.P.: Parallel iteration of high-order Runge-Kutta methods with stepsize control. J. Comput. Appl. Math. **29** (1990) 111–127
4. Prince, P.J., Dormand, J.R.: High order embedded Runge-Kutta formulae. J. Comput. Appl. Math. **7**(1) (1981) 67–75
5. Korch, M., Rauber, T.: Scalable parallel RK solvers for ODEs derived by the method of lines. In: Euro-Par 2003. Parallel Processing. LNCS 2790, Springer (2003) 830–839
6. Korch, M., Rauber, T.: Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. J. Par. Distr. Comp. **6**(3) (2006) 444–468
7. Jackson, K.R., Nørsett, S.P.: The potential for parallelism in Runge-Kutta methods. Part 1: RK formulas in standard form. SIAM J. Numer. Anal. **32**(1) (1995) 49–82
8. Hoffmann, R., Korch, M., Rauber, T.: Performance evaluation of task pools based on hardware synchronization. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2004) 44
9. Korch, M., Rauber, T.: A comparison of task pools for dynamic load balancing of irregular algorithms. Concurrency and Computation: Practice and Experience **16** (2004) 1–47
10. Lioen, W.M., de Swart, J.J.B.: Test Set for Initial Value Problem Solvers, Release 2.1. CWI, Amsterdam, The Netherlands. (1999)
11. Hussels, H.G.: Schrittweitensteuerung bei der Integration gewöhnlicher Differentialgleichungen mit Extrapolationsverfahren. Diploma thesis, University of Cologne, Cologne, Germany (1973)
12. Lecar, M.: Comparison of eleven numerical integrations of the same gravitational 25-body problem. Bulletin Astronomique **3** (1968) 91
13. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I: Nonstiff Problems. 2nd rev. edn. Springer, Berlin (2000)
14. Banicescu, I., Carino, R., Pabico, J., Balasubramaniam, M.: Design and implementation of a novel dynamic load balancing library for cluster computing. Parallel Computing **31**(7) (2005) 736–756
15. Tabirca, S., Tabirca, T., Yang, L.T., Freeman, L.: Evaluation of the feedback guided dynamic loop scheduling (FGDLS) algorithms. IEICE Trans. Inf. & Syst. **E87-D**(7) (2004) 1829–1833
16. Nieh, J., Levoy, M.: Volume rendering on scalable shared-memory MIMD architectures. In: Proceedings of the Boston Workshop on Volume Visualization, ACM Press (1992) 17–24