Proc. ARCS'06 Workshop on Parallel Systems and Algorithms (PASA'06), Frankfurt (Main), Germany, March 2006.

Lecture Notes in Informatics (LNI), vol. P-81, pp. 313–322, ISBN 3-88579-175-7. GI, 2006.

Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments

Mattias Eriksson, Christoph Kessler, Mikhail Chalabine
PELAB, Institute for Computer and Information Science,
Linköping university, Sweden
{ x05mater, chrke, mikch } @ ida.liu.se

Abstract: We study strategies for local load balancing of irregular parallel divide-and-conquer algorithms such as Quicksort and Quickhull in SPMD-parallel environments such as MPI and Fork that allow to exploit nested parallelism by dynamic group splitting. We propose two new local strategies, repivoting and serialisation, and develop a hybrid local load balancing strategy, which is calibrated by parameters that are derived off-line from a dynamic programming optimisation. While the approach is generic, we have implemented and evaluated our method for two very different parallel platforms. We found that our local strategy is superior to global dynamic load balancing on a Linux cluster, while the latter performs better on a tightly synchronised shared-memory platform with nonblocking, cheap task queue access.

1 Introduction

Technical limits on processor clock rates and likewise limited exploitable instruction-level parallelism in applications force high-performance computer architectures to increasingly rely on exploiting massive, explicit thread-level parallelism, such as in the form of simultaneous multithreading, multi-core processor structures, and clusters of commodity processors. One possibility to solicit more explicit parallelism in applications consists in exploiting *nested parallelism*, where each task in a parallel computation may, statically or dynamically, spawn more parallel tasks. A somewhat extreme case is recursively nested parallelism as provided by parallel divide-and-conquer algorithms.

Divide-and-conquer (DC) is an important algorithmic problem solving strategy. A problem is split into one or more independent subproblems of smaller size (*divide phase*); each subproblem is solved recursively (or directly if it is trivial) (*conquer phase*), and finally the subsolutions are combined to a solution of the original problem instance (*combine phase*). Examples for DC computations with fixed subproblem sizes (*oblivious DC algorithms*) are mergesort, FFT, or Strassen matrix multiplication; examples for DC computations with runtime-data dependent subproblem sizes (*irregular DC algorithms*) are Quicksort or Quickhull. A classification of DC algorithms is given by Herrmann and Lengauer [7]. In parallel computing, the *parallel DC* strategy allows for a simultaneous solution of all subproblems, as these are independent of each other. However, a significant speedup can generally be obtained only if also the work-intensive parts of the divide and the combine phase can be parallelised.

In parallel programming environments that support the so-called *SPMD* style of execution, such as MPI [5], Tlib [12], Fork [9], or NestStep [10], a fixed number of processors or threads execute the program as a group. Groups can be split dynamically into disjoint subgroups. In parallel DC computations, a parallel solution of subproblems can be achieved by splitting the group of processors assigned to solving a problem instance into several parallel subgroups and recursively solving one subproblem on each subgroup. The implementation switches to the corresponding sequential DC algorithm if the subgroup size reaches one.

Obviously the overall execution time is minimised if the processors of the group can be subdivided in a way that perfectly matches the work distribution for the solution of the corresponding subproblems (*load balancing problem*). While this is easy for oblivious parallel DC algorithms where local load balancing decisions in the divide phase are sufficient, load balancing is a challenge for irregular parallel DC algorithms. In particular, we have to choose between local strategies and global dynamic load balancing mechanisms such as central task queues.

In the parallel divide-phase of a divide-and-conquer algorithm the task is split into subtasks, and the number of processors to assign to each subtask is decided. However assigning processors to tasks leads to imbalance because the number of processors assigned to each group must be an integer larger than zero. For instance, for a group with $P = 8$ processors and a problem that is split into two subproblems where the ratio of expected work is $1.3 : 6.7$, the ideal assignment for the smaller subtask would of course be 1.3 but this is impossible in our group-SPMD environment. Our choices are either one or two processors on the smaller problem; both choices will lead to imbalance between the groups.

The load balancing problem for parallel irregular divide-and-conquer algorithms has been studied before and several solutions to improve performance have been proposed [6, 11]. Most of these solutions let the processor loads become imbalanced and then perform some *active* load balancing in the serial phase of the parallel divide-and-conquer program. One example is to use a task-queue in the serial phase. When the group-size is one and the processor is about to make a recursive call on two non trivial subtasks one of them is put in a shared task-queue. When a processor is finished executing a task and has no more work to do locally it fetches a new task from the queue. The good thing about a task queue is that as long as there are tasks in the queue no processor will be idle. The drawback is that managing a parallel task queue can be costly. A variant of task queue suitable for distributed memory machines is the manager load balancing [6] in which one processor is dedicated to load balancing. The other processors may request help from the manager when they have much work to do in the sequential phase. Overpartitioning [11] is another strategy that reduces imbalance. It works by creating more partitions than there are processors, and when all partitions are made they are assigned to processors. This gives good load balance with high probability. For quicksort there also exists a simplified variant [13] for parallel execution where the data is initially sorted locally on each processor; this presorting makes it possible to select a pivot very close to the median of the data set. Thus the processor group can be split almost exactly in half and each processor will only have to communicate with one other processor. This preconditioning strategy is, however, limited to a specific algorithm.

In this paper, we investigate two new local strategies, *repivoting* and *serialisation*, and develop a *hybrid local load balancing strategy* for irregular parallel divide-and-conquer algorithms that is applicable to SPMD environments. The strategy is calibrated by parameters that are derived off-line by an analysis based on dynamic programming. We have implemented and evaluated our method for two parallel platforms at opposite corners of the spectrum of parallel architectures: the SB-PRAM with tightly synchronised processors and uniform, unit-latency memory access time, using the PRAM programming language Fork [9], and a large Beowulf cluster of Intel Xeon processors, using MPI and the Tlib library for nested parallelism support [12]; our method could be applied to other parallel architectures as well. When comparing the local strategies with global dynamic load balancing, we found that on the SB-PRAM where synchronisation of a shared task queue is cheap and nonblocking, global dynamic load balancing is always the better choice, while on the MPI cluster, our local strategy outperformed global dynamic load balancing.

2 Group splitting, repivoting and serialisation

Using the options group splitting, repivoting, and serialisation, our goal is to choose the strategy that minimises expected execution time.

Group splitting The group splitting strategy uses no active load balancing but simply splits the processor group regardless of imbalance. When deciding how many processors to assign to each subtask we should try to come close to the *ideal assignment*: let $W_1, W_2, \ldots, W_d$ be the expected work loads of the d subtasks and let P be the number of processors in the group. Then we define the *ideal assignment* for subtask k as

$$i_k = \frac{W_k}{\sum_{j=1}^{d} W_j} P$$

The group splitting strategy assigns $j_k = \max\{1, round(i_k)\}$ processors to subtask k where $round(i_k)$ rounds to an integer adjacent to i_k such that $\sum_{k=1}^{d} j_k = P$.

In the following we will, for simplicity, focus on the most common case, $d = 2$.

Repivoting In the parallel divide phase two subtasks are created by a group. When this is done, the sizes of the subtasks are known and it is possible to estimate the workload of each task and thus know what the (local) imbalance will be if the group is split into two independent groups. It is possible to put a threshold on this imbalance. If the imbalance is larger than this threshold, we discard the subproblems and once again go through the process of creating subtasks. This is possible, for example, in quicksort where the created partitions have random size but not in quickhull where the pivot element is determined by the data set.

The tradeoff is obvious: With a low threshold on the imbalance, the load will be well balanced among the processors, but the cost of creating subtasks in the parallel phase will be higher. With a more liberal threshold the imbalance can grow larger but less work will be done in the parallel divide phase.

Serialisation The serialisation strategy will ignore task parallelism between subtasks whenever it would lead to a bad load balance, and instead simply execute the subtasks one after another with the full group. The imbalance can be thresholded as above; when it is larger than the threshold the two subtasks are solved by the full group one after the other, *i.e.*, the group is not split if the split would be bad for load balance. On the other hand, if the imbalance is below the threshold the group splits and the imbalance between processor groups is low.

In this imbalance avoidance strategy the price of serialising is that more of the total work in the task will be executed in the parallel phase. This is bad mainly for two reasons; first, efficiency is lower in the parallel phase because of the synchronisation and communication that is needed. Moreover, for a one-processor machine there often exist sequential algorithms that are better than the parallel ones; *e.g.*, in sequential quicksort, partitioning can be done very efficiently in-place with the median-of-three pivot selection strategy.

Optimal strategy Group splitting, serialisation and repivoting are our three possibilities for local load balancing in the parallel divide phase. We would like to know which of these strategies is the best one on average. To find the optimal strategy, we derive a recurrence relation that describes the running time for an algorithm without any form of active load balancing. Assume that p is the number of processors that executes the algorithm, n is the size of the problem (for example the number of elements to sort in quicksort), $T(n)$ is a stochastic function that describes the running time for the algorithm on one processor, and $D_p(n)$ describes the time needed for p processors to create new subtasks, communicate data and split the processor group in the parallel phase, then

$$T_p(n) = \begin{cases} T(n), & \text{if } p = 1 \\ D_p(n) + \max\{T_{p_1}(n_1), T_{p-p_1}(n_2)\}, & \text{if } p > 1 \end{cases} \tag{1}$$

where p_1, the number of processors assigned to the first subtask, is some function $p_1 = p_1(n_1, n_2)$ with $1 \le p_1 < p$, and $n_1 \in \mathbb{N}$ and $n_2 \in \mathbb{N}$ are stochastic variables whose values describe the sizes of the subtasks.

Now to get to the expected running time, $\mathrm{E}[T_p(n)]$, all possible values of n_1 and n_2 have to be considered. Define $\mathrm{Prob}(n_1 : n_2)$ to be the probability that a task is split into subtasks of sizes n_1 and n_2, with $0 < n_1, n_2 < n$,

$$\mathrm{E}[T_p(n)] = \begin{cases} \mathrm{E}[T(n)], & \text{if } p = 1 \\ \begin{array}{l} \mathrm{E}[D_p(n)] + \displaystyle\sum_{n_1=1}^{n-1} \sum_{n_2=1}^{n-1} \mathrm{Prob}(n_1 : n_2) \cdot \\ \cdot\, \mathrm{E}[\max\{T_{p_1}(n_1), T_{p-p_1}(n_2)\}] \end{array}, & \text{if } p > 1 \end{cases} \tag{2}$$

A simple solution to this equation does not seem to exist [4]. Therefore we solve it numerically, using a discretization of the probability distributions and dynamic programming [2]. The latter allows to calculate optimal strategies for parallel quicksort bottom-up for larger and larger problem sizes and processor groups and store them in a table such that common subinstances are computed only once.

The model of execution for quicksort Let us now look at how dynamic programming can be applied to our case. We are interested in minimising $\mathrm{E}[T_p(n)]$ (see Equation 2). First we note that this expression has two variables, thus we need to store optimal strategies and execution times in a $p \times n$ matrix. This matrix can be created by first filling in the row where $p = 1$, then the row where $p = 2$, and so on.

In order to arrive at the correct optimal decision, we can not simply store the expected value of the execution times, $T_p(n)$, in the matrix because $\mathrm{E}[\max\{T_q(k), T_r(l)\}]$ is not in general equal to $\max\{\mathrm{E}[T_q(k)], \mathrm{E}[T_r(l)]\}$. Hence, we need to store information about the probability distribution of $T_p(n)$. To save computer memory and shorten execution time, this representation of probability distribution can not be allowed to grow huge; therefore we approximate a probability distribution with K pairs $\langle r, t\rangle$ where r is a probability value and t is a value of execution time. In our implementation we used $K = 10$.

The decisions to be optimised are the processor assignment, i.e. how many processors to assign to each subgroup, and if we should use any of the serialisation or repivoting load balancing strategies.

In order to do the necessary calculations we need a model of execution for the intended platform. In Fork we use a simple, empirically supported, model where the expected execution time for one processor is $\mathrm{E}[T(n)] = C_1(2n \ln n + n)$ and the time required for doing partitioning in parallel is $\mathrm{E}[D_p(n)] = C_2(\epsilon \frac{n}{p})$, where ϵ is the efficiency of partitioning in parallel. On our distributed memory machine we found $\mathrm{E}[T(n)] = C_3(2n \ln n + n)$ and $\mathrm{E}[D_p(n)] = C_4 \frac{n}{p} + C_5 n + C_6 p$. See [4] for details about the models.

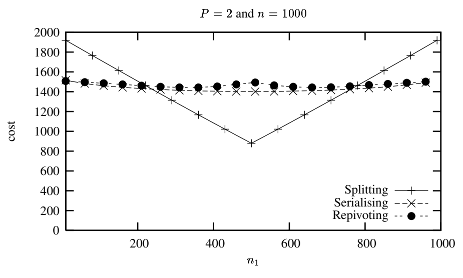
Once the subtask sizes (n_1 and n_2) are known, the expected execution time without load balancing can be calculated as

$$\mathrm{E}[T_p(n)] = \mathrm{E}[D_p(n)] + \mathrm{E}[\max\{T_{p_1}(n_1), T_{p-p_1}(n_2)\}], \tag{3}$$

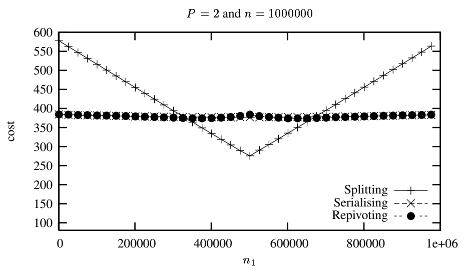
while when serialising the execution the expected execution time will be

$$\mathrm{E}[T_p(n)] = \mathrm{E}[D_p(n)] + \mathrm{E}[T_p(n_1)] + \mathrm{E}[T_p(n_2)]. \tag{4}$$

In both Equations (3) and (4), $\mathrm{E}[T_p(n)]$ can be calculated in the dynamic programming program because we only have to look up values that have already been calculated. However, with repivoting it gets more complicated, as the expected execution time when repivoting depends on values which have not yet been calculated. A solution to this problem is to fill in a full row for a fixed value of p in the matrix, considering only the serialising load balancing strategy, and then check if repivoting had been a better strategy by taking average execution time for all entries in this row. As we will see soon, serialising always has lower expected cost than repivoting, so no values in the row will ever have to be changed.



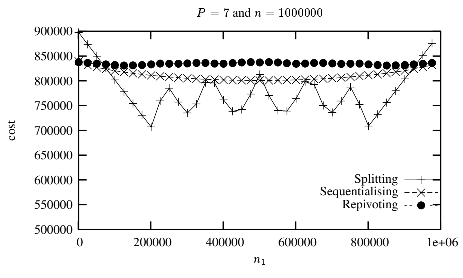
(a)



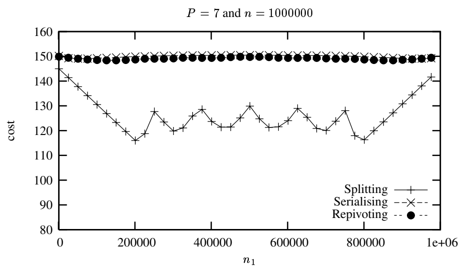
(b)

Figure 1: Plotted in (a) is the expected cost (in ms) to sort 1000 keys with 2 processors on the SB-PRAM, and in (b) the expected cost to sort 1 million elements with 2 processors on the cluster.

Results of dynamic programming Let us now look at the results of the dynamic programming approach. We consider quicksort implemented in both MPI for a distributed memory machine (a Xeon cluster) and in the PRAM programming language Fork and executed on the SB-PRAM simulator [9]. In Figure 1 we can see what the expected cost is when splitting, serialising and repivoting for various sizes of subtask 1 for a fixed problem size n with two processors. We see that the best case is when $n_1 = n/2$, *i.e.* the problem is split in two equally large subproblems. We can also see that, when n_1 is smaller than some value R or larger than some value $n - R$, serialising the execution is the best strategy to use. Important to note is that repivoting is never the best strategy. Actually we have observed in all our examples that repivoting always has expected running time approximately equal or larger than the serialising strategy (see e.g. Figures 1 and 2). We therefore omit repivoting from further consideration.



(a)



(b)

Figure 2: Expected cost (in ms) for 7 processors to sort 1 million elements (a) on the SB-PRAM, and (b) on the distributed memory machine.

Figure 2, where $p = 7$, unveils that the best possible n_1 is not at the $n/2$ mark. Instead the n_1 which gives the lowest expected cost is one where the optimal processor assignment is $1 : 6$. The second dip corresponds to the optimal subtask sizes for splitting the processor group $2 : 5$, and so on. On the edges farthest to the right and farthest to the left, the value is high because one of the processors must be split off and deal with a relatively small

Fork

p **n**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\mathbf{10^2}$	.180	.160	.110	.050	0	0	0	0	0	0	0	0	0	0	0
$\mathbf{10^3}$	.233	.199	.140	.090	.053	.021	0	0	0	0	0	0	0	0	0
$\mathbf{10^4}$	.265	.221	.158	.111	.078	.052	.031	.012	0	0	0	0	0	0	0
$\mathbf{10^5}$	.288	.234	.169	.124	.092	.068	.049	.034	.020	.003	0	0	0	0	0
$\mathbf{10^6}$	.305	.243	.176	.133	.102	.079	.060	.046	.034	.024	.012	0	0	0	0
$\mathbf{10^7}$	.318	.249	.182	.139	.109	.086	.068	.054	.043	.033	.024	.014	.008	0	0
$\mathbf{10^8}$	.329	.254	.187	.144	.113	.091	.074	.060	.049	.040	.032	.024	.015	.015	0
$\mathbf{10^9}$	.338	.258	.191	.147	.117	.095	.078	.064	.053	.045	.037	.030	.024	.015	.016

MPI

p **n**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\mathbf{10^5}$	.156	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\mathbf{10^6}$	.328	.216	.136	.076	.022	0	0	0	0	0	0	0	0	0	0
$\mathbf{10^7}$	.382	.265	.191	.143	.109	.083	.062	.042	.026	.011	0	0	0	0	0
$\mathbf{10^8}$	.401	.281	.206	.159	.126	.102	.083	.067	.054	.043	.033	.025	.015	.008	0
$\mathbf{10^9}$	.407	.284	.210	.163	.131	.107	.088	.073	.061	.051	.042	.034	.027	.021	.016

Table 1: Threshold values extracted from dynamic programming data, for implementations using Fork on the SB-PRAM and using MPI on the Xeon cluster. The numbers indicate that if a subtask is smaller than this number multiplied by problem size, serialisation is the optimal strategy.

subtask that will quickly be solved and this single processor will not be utilised any more for the rest of the task, i.e. processing power is wasted while the larger processor group is overloaded. The first top in the graph corresponds to the subtask sizes where, if processors are assigned $1 : 6$, the single processor will be overloaded and if processors are assigned $2 : 5$ the larger group will be overloaded, i.e. both alternatives are bad ones. We can look at the tops in the graph as corresponding to when pivots are selected such that the processor assignment decision is hardest to make.

All the studied average-cost graphs are symmetric around the line $n_1 = n/2$, as the branching factor in our examples is two and the subtask sizes are related by the equation $n_1 + n_2 = n$ (this is true in quicksort but not in quickhull). Consequently the case with fixed p and n where subtask sizes are n_1 and n_2 is equivalent to the case where subtask sizes are n_2 and n_1.

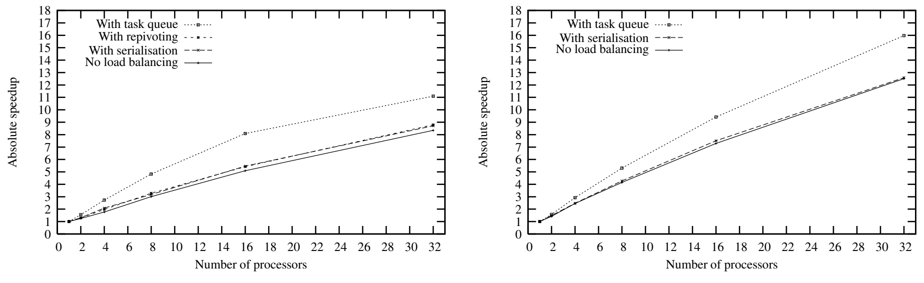
Combined strategy for local load balancing The results from dynamic programming have shown us that serialising the execution is generally worthwhile when one of the subtasks is very small. Now we need to find threshold values on n and p.

In Table 1 we have extracted such threshold values, for a few chosen n and p from dynamic programming data. The table should be read as this: the values represent a ratio, r, such that if one subtask has a size n_i with $\frac{n_i}{n} < r$ then the optimal strategy is to serialise the execution. And otherwise, i.e. if the subtasks are better balanced, the optimal strategy is to split the processor group. For example if $p = 4$, $n = 10000$ and $n_1 = 1000$ we have a situation where $\frac{n_1}{n} = 0.1 < 0.158$ and thus serialising is the best option, while if n_1 had been 2000 then the ratio would have been $\frac{n_1}{n} = 0.2 > 0.158$ and the optimal strategy would be to split the processor group.

3 Comparison of quicksort implementations

We have implemented quicksort in both Fork and MPI with an approximation of the optimal serialisation and repivoting strategies and compared speedups with speedups of quicksort implemented with task queue load balancing.

Figure 3(a) shows speedups when sorting 40000 integers on the SB-PRAM when the pivot is selected as a random element in the parallel phase. We see that repivoting and serialisation are very similar in performance and slightly better than the unbalanced implementation. However the task queue solution is clearly better than both serialisation and repivoting. Figure 3(b) shows speedups of our implementations again, but this time with the better median-of-three pivot selection in the parallel phase. We see again that serialisation is slightly better than using no load balancing at all but the effects of load balancing are lower. Task queue is the best strategy again. Figure 4 shows results of executions on the distributed memory machine. Clearly serialisation is better than using no load balancing. We see that global load balancing using a manager is not very beneficial in our example (we predict that the manager solution would show better results if the number of processors was larger) and that the simplified quicksort version is the best one. It must however be noted that this preconditioning is only possible in quicksort. Thus in other irregular parallel divide-and-conquer algorithms it may be serialisation that is the best load balancing strategy. From the results we draw the following conclusions:



(a) (b)

Figure 3: Average speedup for quicksort on 40000 elements in Fork with different strategies. In (a) a random element is used as pivot in the parallel divide phase and in (b) the pivot is selected as median of 3 random elements.

- For small problem sizes and large number of processors load balancing does not improve the execution time, this is of course because the overhead is higher than what is gained by better balance. In our Fork implementation, 10000 elements on 16 processors is a small problem not worth to load balance.
- Using repivoting and serialisation has similar effects on execution time.
- Load balancing with serialisation is less effective when median-of-3 pivot selection is used, but still has a positive effect.

- On the SB-PRAM the task queue load balancing strategy is superior to the serialisation and repivoting strategies. This is because serialisation only improves balance among processors but imbalance will occur, while with the task queue load imbalance is almost totally eliminated at the cost of maintaining a parallel FIFO queue. The task queue can be implemented extremely efficiently on the SB-PRAM, using the hardware-supported atomic multiprefix operations for non-blocking queue access with almost no overhead. Note that other shared memory platforms may instead require locks or similar mechanisms that sequentialise queue accesses. An evaluation for other shared-memory platforms is an issue for future research.
- On the distributed memory machine our serialisation strategy outperforms the global dynamic manager strategy.
- On the distributed memory machine, both serialisation and manager are beaten by the simplified version of parallel quicksort.

Corresponding results for quickhull can be found in Mattias Eriksson's Master's thesis [4].

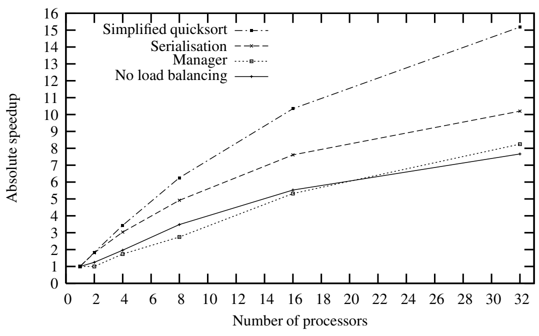


Figure 4: Average speedup for quicksort on 100000000 elements in MPI with different load balancing strategies.

4 Conclusion and future work

We have proposed and studied two new strategies, repivoting and serialisation, for local load balancing of irregular parallel divide-and-conquer algorithms for use in SPMD-parallel environments such as MPI and Fork that allow to exploit nested parallelism by dynamic group splitting. We analysed the optimal local load balancing problem numerically with a dynamic programming method, and developed a hybrid local load balancing strategy. We have implemented our method for two very different parallel platforms, SB-PRAM with Fork and a Xeon Cluster with MPI/Tlib. We found that our local strategy is superior to global dynamic load balancing on the MPI cluster, while global dynamic load balancing performs better on a shared-memory platform with nonblocking, cheap synchronisation. Our technique can likewise be applied to other parallel platforms between these two extremal points of the architecture spectrum, e.g. to SMT-processors, chip multiprocessors and SMP architectures, which is an issue for future work. Moreover, the effects of repivoting and serialisation should be studied for other irregular parallel DC algorithms that divide into more than 2 subtasks.

Acknowledgments We thank the National Supercomputing Center (NSC) at Linköping university for providing access to their Xeon cluster Monolith. We thank Thomas Rauber and his group for providing the Tlib library for recursive parallel task management on top of MPI. — This research was partially funded by SSF RISE and Vinnova GridModelica.

References

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of parallel and distributed computing*, vol 44, nr 1, pp. 71–79, 1997.

[2] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Principles and Practice of Parallel Programming*, pp. 1-12, 1993.

[4] Mattias Eriksson. *Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments*. Master's Thesis, PELAB, Linköpings Universitet, Oct. 2005.

[5] William Gropp and Ewing Lusk and Anthony Skjellum. *Using MPI - 2nd Edition. Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1999.

[6] Jonathan C. Hardwick. An efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms. In *First International Workshop on High-Level Programming Models and Supportive Environments*, pp. 105–114, April 1996.

[7] Christoph A. Herrmann and Christian Lengauer. Parallelisation of Divide-and-Conquer by Translation to Nested Loops. *Journal of Functional Programming* **9**(3):279–310, May 1999.

[8] C. A. R. Hoare. Quicksort. *The Computer Journal* **5**(1):10–16, 1962.

[9] Jörg Keller, Christoph W. Kessler and Jesper Larsson Träff. *Practical PRAM Programming*. Wiley-Interscience, 2001.

[10] Christoph W. Kessler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. *The Journal of Supercomputing* **17**:245–262, Kluwer Academic Publishers, 2000.

[11] Hui Li and Kenneth C. Sevcik. Parallel Sorting by Overpartitioning. In *Proc. 6th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, pp. 46–56, 1994.

[12] Thomas Rauber and Gudula Rünger. Tlib — A Library to Support Programming with Hierarchical Multi-Processor Tasks. *Journal of Parallel and Distr. Computing*, **65**(3):347–360, 2005.

[13] Peter Sanders and Thomas Hansch. On the Efficient Implementation of Massively Parallel Quicksort. *Worksh. on Par. Algorithms for Irregularly Structured Problems*, pp. 13–24, 1997.