Grids and Clusters with Multi-Core Nodes: A Genetics Application Perspective

Mahen Jayawardena[1,2], Henrik Löf[3] and Sverker Holmgren[1]

[1] Division of Scientific Computing, Department of Information Technology, Uppsala University, Sweden
sverker.holmgren@it.uu.se
[2] University of Colombo School of Computing, Colombo, Sri Lanka
mahen@cmb.ac.lk
[3] Department of Energy Resources Engineering, Stanford University, USA
henrik.loef@stanford.edu

Abstract. The introduction of multicore processors imply that algorithms which are parallelized at an outer, coarse grain level should possibly be revisited to examine if multithreading should also be used at an inner, fine grain level.
In this paper we discuss parallel versions of the tightly coupled global optimization algorithm DIRECT. We examine how both coarse grained and fine grained parallelism can be exploited using a hybrid programming model. We show that excellent performance can be archived when using the hybrid algorithm on loosely-coupled systems like clusters and grids with multicore nodes.

1 Introduction

In the construction of microprocessors, the exploitation of instruction-level parallelism has come to an end. Also, we can not pack more transistors into a processor without introducing serious side effects. Further increase in clock frequency will cause too much heat generation. Wire delay is becoming a major problem as the wires get thinner, and the memory systems are not able to handle data access at the same speeds that the processors are.

These problems have recently made the processor architects make a radical shift in how they construct processor chips. Instead of increasing the clock speed and packing more transistors into a single CPU, the industry has moved into multi-core processors. A core is a CPU in it self, and each processor chip contain several such cores. Each core can also have multiple threads executing on it. Presently, most PCs sold have dual-cores processors. The trend is that the number of cores will increase, and that the clock frequency will remain approximately constant, or even be somewhat reduced.

The downside of this development is that much of the software currently in use, which has been written to use just one processor, will often not be able to take immediate advantage of multiple cores. A shift in programming technique is required, where the programmer must become explicitly aware of and utilize the parallelism in the algorithms. Until now the parallelism has been exploited only at the instruction level, and the programmer has usually not been aware of it.

However, programming models which explicitly exploit parallelism have been used by the high performance computing (HPC) community for many years. In HPC, two main classes of parallel architectures are used; Shared memory systems (SMPs) and local memory clusters. In SMPs, multiple processors are connected to a shared memory system. The memory system is normally specially designed, and shared memory servers are often considered to be rather expensive. Clusters are normally built from commodity class nodes, similar to desktop machines, and use some form of interconnect for communication. These systems are often relatively cheap to build, and many HPC systems are of this type. Grid systems can be seen as an extension of clusters, in which a heterogenous collection of machines are connected via the internet and managed via a middleware layer.

Different parallel programming models are normally used for SMPs and clusters. In an SMP, the shared address space provides the possibility of using shared namespace programming models based on multithreading, and the parallelism is often introduced via the use of compiler directives. OpenMP [3] has been a popular choice for programming SMP systems and is widely implemented. OpenMP provides a fork-and-join model where the programm starts with one thread, and when it encounters parallel region it fork threads that work in parallel. An advantage of OpenMP is that it is often rather easy to parallelize an already existing serial code in an incremental way.

For programming local memory clusters the Message Passing Interface (MPI) has become the de-facto standard. Several implementations of MPI are available [1, 8]. In message passing, the processes communicate via calls to library routines that send/receive messages. The message passing model provides full control over the communication and synchronization, but then also demands the user to deal with these issues in detail. Often, it takes some effort to parallelize a serial code.

On a grid system, communication is very expensive and the behavior of the system is rather stochastic. Parallel computations on a grid need to consist of large, disjoint chunks of work that can be performed independent of each other. Often, a data parallel strategy is used where the same computation is independently performed on different sets of data. The communication is set up and handled by the middleware, e.g. via web services.

Today, grids and clusters are built with nodes which have one or more multicore processors. It would be possible to use a message passing programming model that run one process on each core, but this may not result in the most efficient code. A more effective strategy may be to use message passing between nodes and a shared memory model for fine-grained parallelism within a node/multicore processor, e.g. using OpenMP directives. Using this type of programming model may introduce new opportunities for parallelization. Most parallel algorithms are designed to minimize communication, but communication within a multicore chip may in fact be cheaper than a memory reference.

Previously [10] we have parallelized the tightly-coupled global optimization algorithm DIRECT [11] for a specific problem class in genetic analysis. The algorithm is derived by removing some of the global coupling at an outer level, hereby making it suitable for e.g grid systems. However, by modifying the algorithm we also change its

performance. Using too many parallel processes results in that the total work needed is increased, and the speedup is reduced. The aim of this paper is to examine if a more fine-grained parallelism where the global coupling is retained can also be included in the algorithm. This second level of parallelism is implemented using multithreading, e.g. OpenMP, resulting in a hybrid message-passing/OpenMP code which can possibly be highly suitable for clusters with multicore nodes.

The rest of this paper is organized as follows: We start by describing the genetics application in Section 2. The parallelism available in DIRECT is discussed in Section 3, and the computer systems used are described in section 4. In Section 5, we present the results of investigations of the performance of the new algorithm, and some conclusions and future work are discussed in Section 6.

2 Application: Genetic Analysis of Quantitative Traits

Traits or characteristics such as body weight, susceptibility to cancer and heart disease in humans, growth rate in farm animals, and crop yield in plants are all controlled by a combination of genetic factors and the environment. These type of traits are called *quantitative*, since they vary continuously as opposed to mendelian or single gene traits such as blood group. By using a QTL mapping software that performs a statistical analysis of genetic and phenotypic information for a large population, *quantitative trait loci* (QTL), i.e. loci in the genome affecting the trait, can be identified. A review of QTL analysis is given e.g. in [7, 14].

The QTL mapping code used in this paper employs a model where d QTL affect the trait under study. A set of d locations in the genome can be identified by the vector $\mathbf{x} = [x_1 \quad x_2 \ldots x_d]$, where $x_i \in [0, G]$ and G is the size of the genome. Let n be the number of individuals in the population, $\mathbf{y}$ the vector of n phenotype observations, $k \geq d$ the total number of parameters in the model, and $\mathbf{b}$ the vector of k effects. A general linear QTL model can then be formulated in matrix form,

$$\mathbf{y} = A(\mathbf{x})\mathbf{b} + \epsilon, \tag{1}$$

where the $n \times k$-matrix $A(\mathbf{x})$ is the design matrix and ϵ is the error vector. In the QTL mapping procedure, this model is evaluated for different combinations of locations **x** and the best model fit is searched for. This corresponds to solving the optimization problem

$$RSS_{opt} = \min_{\mathbf{b},\mathbf{x}} (A(\mathbf{x})\mathbf{b} - \mathbf{y})^T (A(\mathbf{x})\mathbf{b} - \mathbf{y}). \tag{2}$$

The problem (2) can be separated into two parts. The first, inner problem is to compute the model fit for a given position **x**, referred to as the objective function evaluation (3). The second, outer problem is the global search (4).

$$RSS(\mathbf{x}) = \min_{\mathbf{b}} (A(\mathbf{x})\mathbf{b} - \mathbf{y})^T (A(\mathbf{x})\mathbf{b} - \mathbf{y}), \tag{3}$$

$$RSS_{opt} = \min_{\mathbf{x}} RSS(\mathbf{x}), \tag{4}$$

The global search is a d-dimensional global optimization problem. Figure 1 shows a part of a typical optimization landscape for a problem with $d = 2$. When searching for d QTL, the search space is in principle a d-dimensional hypercube where the side is G.

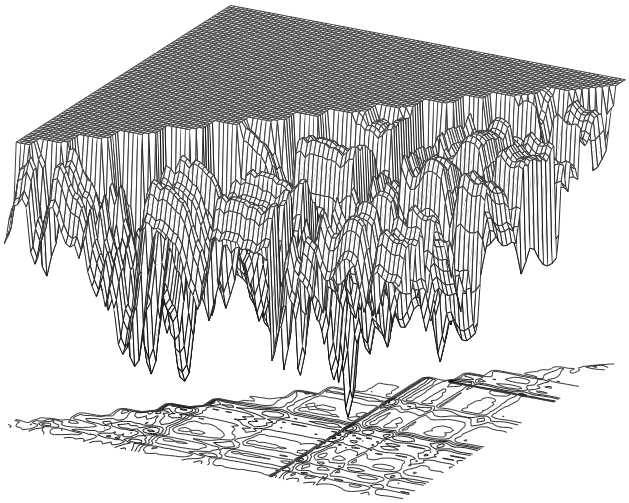

Fig. 1. Part of a typical optimization landscape for a model where $d = 2$

The standard QTL mapping softwares [5, 6, 17] solve the global search problem by using the brute-force exhaustive search algorithm. Because of the computational constraints, these type of algorithms/codes can not be used to search for more than two QTL simultaneously. In [12] and [13], a more efficient way of performing the search is demonstrated, using a slightly modified version of the global optimization algorithm DIRECT [11].

DIRECT works by recursively identifying hyper-rectangles in the search space where the global optimum is likely to be found, and then sub-dividing these hyper-rectangles into smaller ones. In the original DIRECT scheme the search starts by evaluating the objective function at the mid-point of the search space. For the global search problem in QTL mapping, we instead use a multi-point initialization where the natural partitioning of the genome into c chromosomes to divide the d-dimensional hypercube search space into a set of c^d d-dimensional chromosome combination boxes, *cc boxes*. The objective function is discontinuous at the cc-box boundaries, but by using the multi-point initialization we can retain the theoretical results available for DIRECT

when the objective function is Lipschitz-continuous. This type of initialization scheme also provides some additional parallelism, as will be discussed in Section 3.

DIRECT works in an iterative manner, and in each iteration the hyper-rectangles are classified based on their size. In the implementation we use, the hyper-rectangles are clustered based on their longest side. Within each class, the hyper-rectangles are again sorted based on the objective function values. Different datastructures can be used to store the sorted hyper-rectangles. This is further discussed in Section 3, and some alternative choices are discussed in Section 6. By plotting the class size of the hyper-rectangles versus the objective function values and determining the lower convex hull of the graph, the ”potentially optimal” hyper-rectangles are identified. These hyper-rectangles are then subdivided, and the next iteration is started. In our implementation, hyper-rectangles smaller than a given size will not be subdivided further, and our algorithm will eventually perform an exhaustive search on a mesh given by this resolution. However, to take advantage of DIRECTs ability to normally locate the optimium much earlier in the process, the iterations should be stopped using some other criterion. In our implementation, a given number of function values N_f are allowed without further improvement of the global minimum. In Table 1, the values of N_f used are given. The table also lists the number of cc-boxes in the search space. More details on how DIRECT is adopted to the global search problem in QTL analysis are given in [12, 13].

Table 1. Stopping criterion parameters. The minimum number of function evaluations allowed without improvement of the global minimum (N_f)

d	N_f	cc-boxes	Objective Function Evaluations
2	841	171	2143
3	11487	1140	58390
4	117740	5985	549116
5	965468	26334	4569085
6	6597367	100947	-

3 Parallelization of DIRECT for the QTL Mapping Application

DIRECT is an inherently tightly-coupled algorithm. There is no apparent, easy-accessible parallelism available. However, parallelism *can* be introduced into the algorithm at several levels: The search space can be partitioned and each processor can work on one of these partitions. We denote this strategy ’Partitioned search space’. The second level of parallelism is found in the evaluation of the objective function for the potentially optimal hyper-rectangles. Once these are identified, these evaluations can be done in parallel. We denote this strategy ’Parallelized convex hull’. The innermost level of parallelism may be available within the evaluation of the objective function. This strategy is denoted ’Parallelized objective function’.

In [9], a version of DIRECT which exploits the different levels of parallelism described above is discussed. Here, the original algorithm is parallelized and the tight coupling imposed by the central datastructures is retained. The implementation in [9] targets tightly-coupled clusters and other systems which can support the extensive communication needed. In [10] we present another parallelized version of DIRECT, based on the strategy of partitioning the search space. In this case, some of the tight-coupling of the algorithm is removed, resulting in that the algorithm itself is modified. The implementation presented in [10] targets loosely-coupled systems and grids, and we show that this version of DIRECT can be successfully applied for solving QTL mapping problems on such systems.

We now discuss the different levels of parallelism in the DIRECT algorithm when applied to the global search problem in QTL mapping.

3.1 Partitioned search space

The approach used in [10] is to partition the search space into sets of cc-boxes and then let the processors/jobs independently apply the DIRECT scheme to one such set each. This effectively means that each job finds the local minimum for it's cc-box set. Once these local minima are computed and transferred to the host job, a trivial serial search give us the global minimum. The motivation for partitioning the search space is this manner and removing the transfer of data between the cc-box sets in the DIRECT algorithm is that the objective function (3) is discontinuous at the cc-box boundaries, and the search can in a sense be viewed as many independent problems.

The advantage of this strategy is that jobs/processors do not need to communicate with each other. This is a requirement if we plan to use the algorithm in a grid environment. Much of the potential speed-up for this algorithm originates from that when we reduce the size of the local search space we should also be able to reduce the number of function evaluations in the termination criterion. In (5), the formula that is used for this reduction is given.

$$N_f(p) = N_f \cdot a^{\log_2(p)}, \tag{5}$$

The optimal value for a is $a = 0.5$, but for some problems a slightly larger value must be used. Also, it should be noted that this type of parallelization will normally introduce load imbalance. The data is statically partitioned, and it is not known a-priori how many evaluations of the objective function that will be performed by each processor/job. We have partitioned the cc-boxes in a block-cyclic way, trying to improve the load balance as far as possible. The initialization phase of each processor requires the same number of function evaluations, and in the hybrid code this part of the parallelism is implemented using OpenMP.

3.2 Parallelized convex hull

The main data structure in DIRECT is the two-dimensional structure needed to store the clusters of hyper-rectangles where the objective function has been evaluated. The

most flexible implementations use a two-dimensional priority queue which can then be seen as a sorted, dynamic array. The different hyper-rectangle classes are stored in one queue and for each entry in this queue there is another queue to store information about the individual hyper-rectangles. Various algorithmic aspects need to be consider to provide fast insertions and sorting. Provided that we have some kind of estimate for the number of subdivisions we may use more static data structures which may simplify the implementation. In our implementation, we have used a linked list for the different clusters which is sorted on the size of the hyper-rectangle classes. Another linked list is used to store the individual hyper-rectangles within a given class, and this list sorted on the objective function value.

We have parallelized the evaluation of the objective function for the potentially optimal hyper-rectangles in the convex hull. At each DIRECT iteration we store references to the potentially optimal hyper-rectangles in an array. We then add OpenMP work-share directives to the loop that indexes the array. The amount of parallelism is limited by the number of potentially optimal hyper-rectangles. Also, shared data structures are accessed, and these accesses must be protected. For example, the insertion of new nodes in the linked lists is performed in critical sections in the code. This serial execution causes some loss of efficiency. In section 6 some possible improvements are discussed.

3.3 Parallelized objective function

The evaluation of the objective function for the QTL mapping problem consists of solving a least squares problem. This is performed using calls to the LAPACK [2] library. Since the objective function is called in parallel, the LAPACK implementation needs to be thread-safe. If the runtime system allows for nested parallelism then the LAPACK routines themselves can be parallelized. For the experiments presented below we have not used a parallelized version of the library.

4 Computer systems

For testing the new hybrid code we have used the Isis cluster at the Uppsala University HPC center UPPMAX [4]. Isis contains 200 nodes where each node consists of two dual-core processors. Each core is a 64-bit AMD opteron @ 2.4 GHz. Isis has a total memory of 1600 GB and uses switched Gigabit ethernet as the interconnect. The system runs Linux, and the jobs are submitted using the GridEngine N1 queuing system.

For testing the new OpenMP parallelization in our code we use a SunFire 15k system at UPPMAX. The partition of the system used for our experiments consists of 32 UltraSPARC III+ CPUs running at 900 MHz, all sharing 48 GB primary memory. The system runs Solaris 9.

5 Results

For the experiments with the parallelized versions of DIRECT we use an experimental data set from two F_2 crosses between outbred lines of wild boars and domestic pigs. The data is determined from 191 individuals, and the genome size is 2300cM.

We begin by describing the results for pure OpenMP parallelization, no partitioning of the search space has been made. The experiments are performed on the SunFire 15k system.

5.1 Only OpenMP parallelization

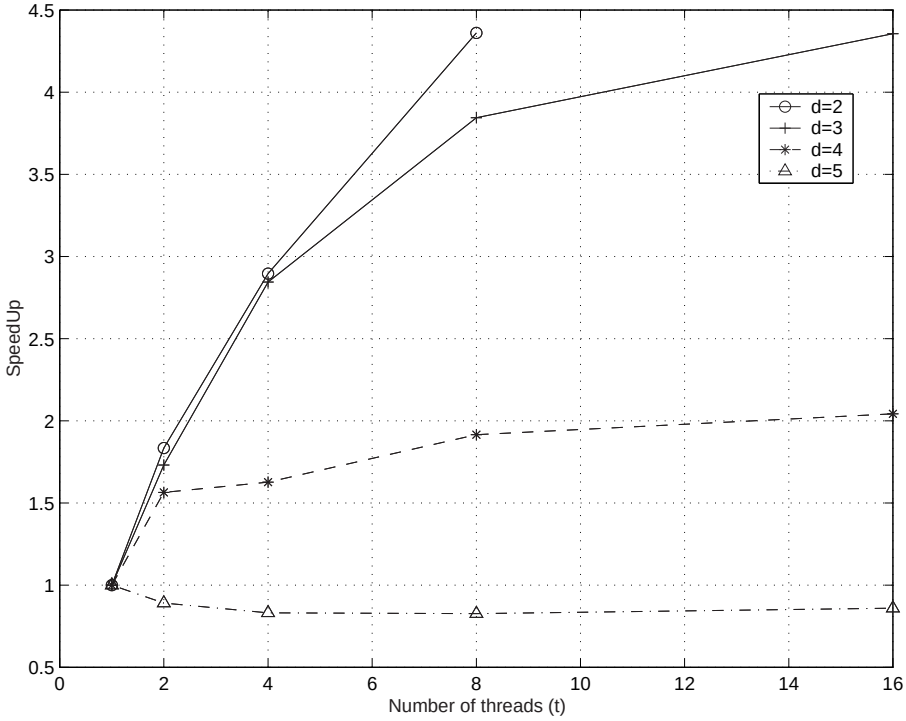

Fig. 2. OpenMP results on Sun System

Figure 2 shows the speedup when using only shared memory parallelization. For $d = 2$, the speedup is reasonable, but as d increases the speedup decreases. As can be seen from table 1, as the dimension increase so does the number of cc-boxes and the total number of function evaluations. The work for updating the data structures in

Table 2. profile of serial code

Dimension (d)	Objective function evaluations	
	Initialization	Search loop
2	7.39%	92.61%
3	1.91%	98.09%
4	1.08%	98.92%

the critical sections becomes a bottleneck when the number of function evaluations is large.

As explained in section 3, the multithreading is used in two places in the code. This first is in the initialization, where the objective function is evaluated for each cc-box in the cc-box set (which consists of all cc-boxes for the experiments described above). This parallelization is at the ”Partitioning the search space”-level, and the parallelism is high. Then, multithreading is used in the main search loop, where the function evaluations for boxes in the convex hull are parallelized. The level of parallelism in the initialization section depends on the number of cc-boxes. In table 2, the fractions of the number of objective function evaluations in the two phases of the code is given. It is clear that only a fraction of the total run time is actually used for the initialization.

The level of parallelism available in the main search loop depend on the number of boxes in the convex-hull. In Table 2), we can also see that the time spent in this phase of the code increases with the dimension. In figures 3, 4 and 5 we show how many hyper-rectangles that are identified as potentially optimal in each iteration, i.e. how many threads that can be used efficiently in this phase of the code. It is clear that the number increases slightly with the dimension, and for 4 and 5 dimensions on average about 10 threads can be utilized. Additional threads will basically be idle.

5.2 Both Data Partitioning and OpenMP Parallelization

Table 3. Load balance

Jobs (p)	d=3			d=4			d=5		
	t=1	t=2	t=4	t=1	t=2	t=3	t=1	t=2	t=3
1	1	1	1	1	1	1	1	1	1
2	1	1	1.111	1.558	1.642	1.699	1.665	1.705	1.692
4	1.143	1.066	1.2	1.484	1.558	1.710	1.589	1.553	1.639
8	1.103	1.263	1.231	1.635	1.646	1.717	1.949	2.1	2.448
16	-	-	-	1.714	1.823	1.969	2.395	2.696	2.671
32	-	-	-	1.789	1.924	2.028	3.501	3.833	2.671
64	-	-	-	-	-	-	2.592	3.006	3.161

Table 4. Runtime (s)

Jobs (p)	d=3			d=4			d=5		
	t=1	t=2	t=4	t=1	t=2	t=3	t=1	t=2	t=3
1	27	17	11	927	900	853	78427	83933	76305
2	12	7	5	272	229	198	23148	22408	22419
4	8	4	3	193	143	127	11002	10866	11054
8	4	3	2	91	57	41	5755	6651	6199
16	-	-	-	69	45	32	2399	2556	2446
32	-	-	-	43	27	18	1748	1812	1703
64	-	-	-	-	-	-	653	627	562

In these experiments, we use both parallelization strategies of dividing the search space and parallelizing the convex hull calculations. The experiments are performed using the cluster Isis, but the parallel implementation can easily be modified also for other loosely-coupled systems, e.g. a grid. The parallel execution is implemented as a two step process. First, a small separate code is used to partition the global search space into sets of cc-boxes. The partitioning code stores the cc-box-set data on a separate file for each set, and these files are then used as input for the DIRECT code run in each of the p computational jobs. A job manager script runs the partitioning code and submits the DIRECT jobs together with the description files, waits for all the jobs to finish, gets the local results, and finally computes the final output.

Since we use a partitioned data set we use equation (5) to determine the number of function evaluations for the stopping rule. In [10] we have evaluated equation (5) as a replacement for a parallel DIRECT version. The value a = 0.75 is used for the current experiments.

The speedup and run times are given in figures 6 to 8 and Table 4. In several cases we can see superlinear speedup. The main reason for this is that in the serial code, DIRECT gets stuck in local minima for some time. When the search space is partitioned, these minima are found by different jobs and the total number of objective function evaluations is reduced. The serial performance is in fact improved by introducing the parallel algorithm.

For $d = 3$, the timing values given in Table 4 are just a couple of seconds, and no proper pattern can be seen in Figure 6. In Figure 7, nice speedup can be seen. Here, multithreading helps to increase the performance that is already provided by the partitioning the search space. In Figure 8, some of the multithreading bottlenecks described in section 3 begin to be apparent.

The load balance is accounted for in Table 3. For $d = 3, 4$ The load balance is quite good. But for $d = 5$ the load balance is a bit worse, and this is also partially explains the behavior shown in Figure 8.

When using the partitioning of the search space in a grid architecture, a baby-sitting application can be employed to perform the job submission and then the download of results after execution. Such small codes are usually coded in Python. A grid portal [18] can also provide a GUI front end for this type of code. On a cluster, another alternative

is to use MPI for the communication. Rather than having input files that describe the cc-box sets a master-slave MPI code could handle this.

5.3 Should we use more search space partitioning or more multithreading?

In Tables 5, 6 and 7 we investigate how best to utilize 4,8 or 16 cores on Isis for a three-dimensional search. It is clear that in all cases investigated, a combination of the two techniques gives the best performance.

Table 5. How to use 4 available processors for 3D

Threads (t)	Jobs (p)	Runtimes (s)
1	4	30.579647
2	2	28.334369
4	1	40.17063

Table 6. How to use 8 available processors for 3D

Threads (t)	Jobs (p)	Runtimes (s)
1	8	19.504535
2	4	16.782802
4	2	16.823821
8	1	30.899

Table 7. How to use 16 available processors for 3D

Threads (t)	Jobs (p)	Runtimes (s)
1	16	23.151731
2	8	10.872763
4	4	10.166195
8	2	12.108945
16	1	28.84

6 Conclusion and future work

In this paper, we have studied parallel versions of the global optimization algorithm DIRECT, applied to a search problem in genetic analysis. We have evaluated the possibility of using both coarse grained parallelism using a partitioning of the search space and fine grained parallelism for performing certain computations inside the DIRECT scheme. The outer level parallelism is implemented using a job submission script which can easily be modified for different loosely coupled architectures, e.g. grids. The inner level parallelism is implemented using multithreading in OpenMP. The aim is to develop a flexible and efficient implementation which is suitable for grids and clusters with multicore nodes.

A main result of the paper is that, by combining the two forms of parallelism, we arrive at a parallel implementation which has superlinear speedup compared to the standard serial implementation.

However, we also find that at the inner level, the parallel efficiency is hampered by the need to update the central, shared data structures in critical regions. One of the problems was when we need to insert new nodes in the lists of hyper-rectangles where the objective function is evaluated. Presently, linked lists are used to store the nodes. A better alternative could be to use a balanced binary trees.

Also, it is possible for different threads to access one of the main lists without any concurrency issues. This can be implemented via the use of locks. Also it is possible to extend this with more efficient locking methods such as queue-based locks such as the MCS [16] lock or CLH lock [15] or we can use lock- free datastructures using atomic compare and swaps directly. The first linked list could then be implemented as a priority queue [19] with the second list implemented as an balanced binary tree. This would eliminate the necessity for any locking mechanism.

References

1. LAM/MPI. www.lam-mpi.org.
2. LAPACK. www.netlib.org/lapack.
3. OpenMP. www.openmp.org.
4. Uppsala multidisciplinary center for advanced computational science (uppmax). www.uppmax.uu.se.
5. C. Basten, B. Weir, and Z.-B. Zeng. *QTL Cartographer, version 1.15*. Dept. of Statistics, North Carolina State University, Raleigh, NC, 2001.
6. K.W. Broman, H. Wu, S. Sen, and G.A. Churchill. R/qtl mapping in experimental crosses. *Bioinformatics*, 19(7):889–890, 2003.
7. R.W. Doerge. Mapping and analysis of quantitative trait loci in experimental populations. *Nature reviews-genetics*, 3:43–52, 2002.
8. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
9. J. He, A. Verstak, L.T. Watson, and M. Sosonkina. Design and implementation of a massively parallel version of DIRECT. Technical report, Technical Report TR-06-02, Computer Science, Virginia Tech., 2006.
10. M. Jayawardena and S. Holmgren. Grid-enabling an efficient algorithm for demanding global optimization problems in genetic analysis. Technical report, Division of Scientific Computing,Dept of IT, Uppsala University, 2007.
11. D. Jones, C. Perttunen, and B. Stuckman. Lipschitzian optimization without the lipschitz constant. *J. Optimization Theory App*, 79:157–181, 1993.
12. K. Ljungberg, S. Holmgren, and Ö. Carlborg. Simultaneous search for multiple QTL using the global optimization algorithm DIRECT. *Bioinformatics*, 20:1887–1895, 2004.
13. K. Ljungberg, M. Kateryna, and S. Holmgren. Efficient algorithms for multi-dimensional global optimization in genetic mapping of complex traits. Technical report, Division of Scientific Computing,Dept of IT, Uppsala University, 2005.

14. M. Lynch and B. Walsh. *Genetics and Analysis of Quantitative Traits*. Sinauer Associates, Inc., 1998.
15. P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *8th International Symposium on Parallel Processing (IPPS)*, pages 165–171, 1994.
16. J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
17. G. Seaton, C. Haley, S. Knott, M. Kearsey, and P. Visscher. QTL express: mapping quantitative trait loci in simple and complex pedigrees. *Bioinformatics*, 18:339–340, 2002.
18. S. Toor, M. Jayawardena, J. Lindemann, and S. Holmgren. A grid portal implementation for genetic mapping of multiple QTL. Preliminary Technical Report, Department of Information Technology, Uppsala University, 2007.
19. N. Scherer. William III, Doug. Lea, and Michael. L. Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147 – 156, 2006.

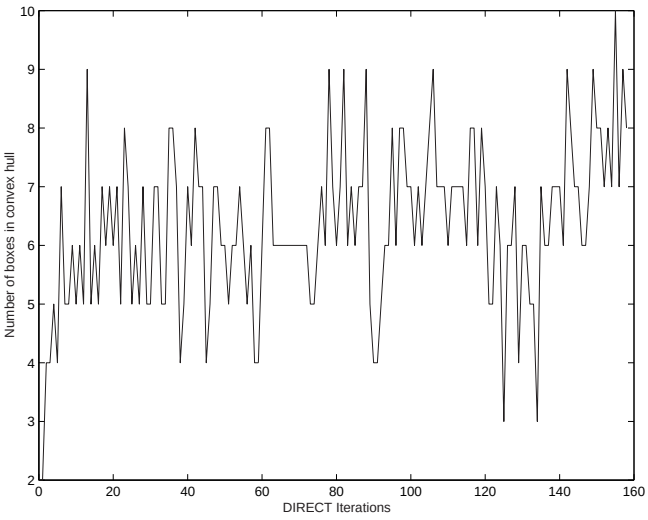

Fig. 3. Level of possible parallelism in search loop for 2D

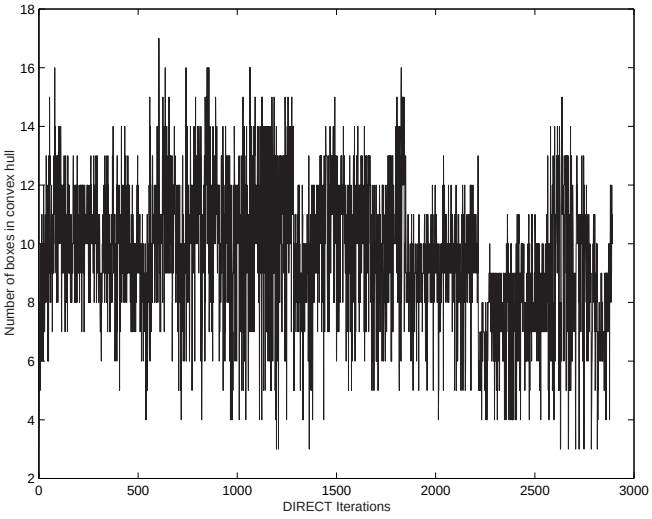

Fig. 4. Level of possible parallelism in search loop for 3D

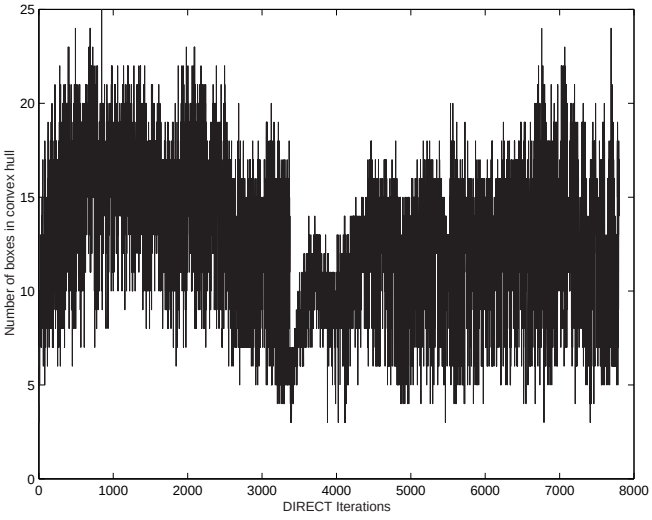

Fig. 5. Level of possible parallelism in search loop for 4D

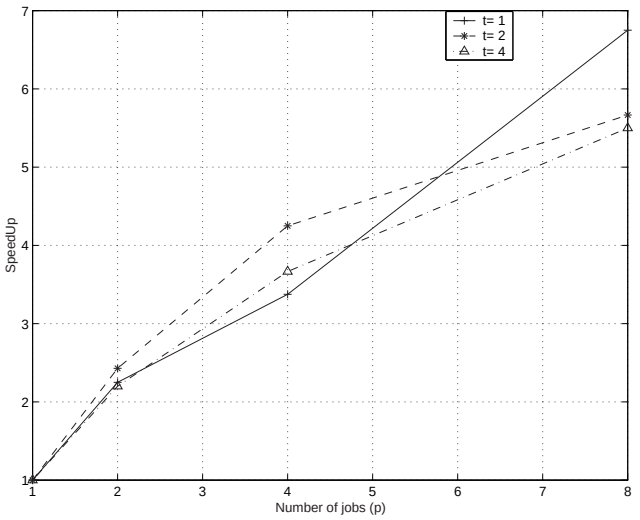

Fig. 6. SpeedUp for data partitioning and multithreading - 3D

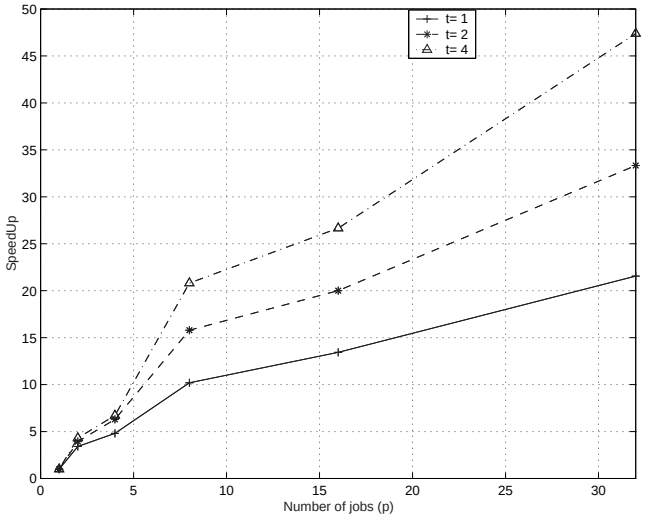

Fig. 7. SpeedUp for data partitioning and multithreading - 4D

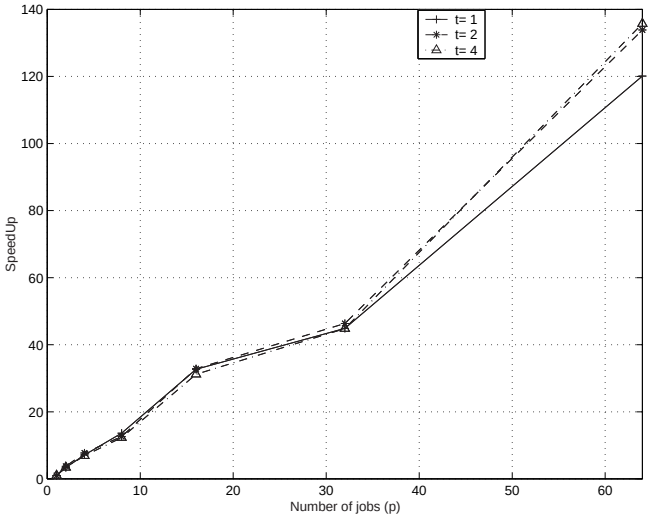

Fig. 8. SpeedUp for data partitioning and multithreading - 5D