Novel parallelization of Simulated Annealing and Hooke & Jeeves search algorithms for multicore systems with application to complex fisheries stock assessment models

Sergio Vázquez[a], María J. Martín[a,*], Basilio B. Fraguela[a], Andrés Gómez[b], Aurelio Rodríguez[b], Bjarki Þór Elvarsson[c]

[a] *Grupo de Arquitectura de Computadores*
Universidade da Coruña, Spain
[b] *Galicia Supercomputing Center (CESGA)*
Santiago de Compostela, Spain
[c] *Marine Research Institute*
Reykjavik, Iceland

Abstract

Estimating parameters of a statistical fisheries assessment model typically involves a comparison of disparate datasets to a forward simulation model through a likelihood function. In all but trivial cases the estimations of these models tends to be time-consuming due to issues related to multi-modality and non-linearity. This paper develops novel parallel implementations of popular search algorithms, applicable to expensive function calls typically encountered in fisheries stock assessment. It proposes two versions of both Simulated Annealing and Hooke & Jeeves optimization algorithms with the aim of fully utilizing the processing power of common multicore systems. The proposals have been tested on a 24-core server using three different input models. Results indicate that the parallel versions are able to take advantage of available resources without sacrificing the quality of the solution.

Keywords: Marine Ecosystems, Gadget, Simulated Annealing, Hooke & Jeeves, Multicore Systems, OpenMP

1. Introduction

Statistical fisheries models typically involve a comparison of the output of a non-linear model of fish population dynamics with available data through a likelihood function. The choice of an optimization algorithm for these types of model can be far from trivial. In all but the simplest examples, such as stock production models [1], where the data required to contrast with the model is relatively small, the time required for adequate parameter estimation can be substantial and estimation issues, such multi-modal likelihoods, can increase the complexity further [see 2, and references therein].

Commonly the parameter estimation procedure involves a combination of search algorithms, both global and local, in an attempt to combine the strengths of a global search with the speed of a local search algorithm. Various combinations of search algorithms have been investigated to identify an opti-

*Corresponding author
Email addresses: sergio.vazquez@udc.es (Sergio Vázquez), mariam@udc.es (María J. Martín), basilio.fraguela@udc.es (Basilio B. Fraguela), agomez@cesga.es (Andrés Gómez), aurelio@cesga.es (Aurelio Rodríguez), bthe@hafro.is (Bjarki Þór Elvarsson)

mal search procedure for a specific task [e.g. 3, 4], but these investigations indicate that a particular combination is problem specific.

Various tools have been developed to aid in the creation of statistical stock assessment models. One such tool is Gadget (Globally applicable Area Disaggregated General Ecosystem Toolbox), which is a modeling environment designed to build models for marine ecosystems, including both the impact of the interactions between species and the impact of fisheries harvesting the species [5, 6]. It is an open source program written in C++ and it is freely available from the Gadget development repository at `www.github.com/hafro/gadget`.

Gadget works by running an internal model based on many parameters describing the ecosystem, and then comparing the output from this model to observed measurements to obtain a goodness-of-fit likelihood score. By using one or several search algorithms it is possible to find a set of parameter values that gives the lowest likelihood score, and thus better describe the modeled ecosystem. The optimization process is the most computationally intensive part of the process as it commonly involves repeated evaluations of the computationally expensive likelihood function, as the function calls a full ecosystem simulation for comparison to data. In addition to that, multiple optimization cycles are sometimes performed to ensure that the model has converged to an optimum as well as to provide opportunities to escape from a local minimum, using heuristics such as those described by [7]. Once the parameters have been estimated, one can use the model to make predictions on the future evolution of fish stocks.

Gadget can be used to assess a single fish stock or to build complex multi-species models. It has been applied in many ecosystems such as the Icelandic continental shelf area for the cod and redfish stocks [7, 8, 9], the Bay of Biscay in Spain to predict the evolution of the anchovy stock [10], the North East Atlantic to model the porbeagle shark stock [11], or the Barents Sea to model dynamic species interactions [12]. Models developed using the Gadget framework have also been used to provide tactical advice on sustainable exploitation levels of a particular resource. Notably the southern European Hake stock, Icelandic Golden redfish, Tusk and Ling stocks, and Barent sea Greenland halibut are formally assessed by the International Council for the Exploration of the Sea (ICES) using Gadget [see 13, 14, 15, 16, for further details].

The aim of this work is to speedup the costly optimization process of Gadget so that more optimization cycles can be performed and a more reliable model can be achieved. The methodology followed comprises three main stages: First, the code was profiled in order to identify its bottlenecks; then, sequential optimizations were applied to reduce these bottlenecks; finally, the most used optimization algorithms were parallelized.

Currently, there are three different optimization algorithms implemented in Gadget: the Simulated Annealing (SA) algorithm [17], the Hooke & Jeeves (H&J) algorithm [18], and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [19]. All of them can be used alone or combined into a single hybrid algorithm. Combining the wide-area searching characteristics of SA with the fast local convergence of H&J is, in general, an effective approach to find an optimum solution. In this work both algorithms have been parallelized using OpenMP [20], so that the proposed solution can take advantage of today's ubiquitous multicore systems.

The rest of this paper is organized as follows. Section 2 covers related work. Section 3 describes the sequential optimizations applied. Sections 4 and 5 describe the basics of the SA and the H&J search algorithms and discuss the parallelization strategies proposed for each one of them. Section 6 presents the experimental evaluation of all the proposals on a multicore system. Finally, conclusions are discussed in Section 7.

2. Related Work

There exist in the literature a large number of solutions for the parallel implementation of the SA algorithm on distributed memory systems [21, 22, 23, 24, 25] In [24] five different parallel versions are implemented and compared. The main difference among them is the number of communications required and the quality of the solution. The solution is better

when the communication among the processes increases. However, in distributed memory systems the communication overhead plays a important role in the global performance of the parallel algorithm. Thus, solutions where communication takes place after every move are not considered a good alternative on these architectures. In shared-memory systems, however, this kind of solution is viable in terms of computational efficiency, as in this case the communications are performed through the use of shared variables.

Parallelization proposals for shared-memory systems are much more scarce. In [26] A. Bevilacqua implements an adaptive parallel algorithm for a non-dedicated shared memory machine using the POSIX Pthreads [27] library. The program execution flow switches dynamically at run-time between the sequential and the parallel algorithm depending on the acceptance ratio and the workload conditions of the machine.

Paramin [28, 6] is a parallel version of Gadget originally written with PVM [29], and later migrated to MPI [30]. As the parallel version proposed in this paper, Paramin is based on the distribution of the function evaluations among the different processes. However, unlike our approach, in each parallel step it updates all the parameters whose moves lead to a better likelihood value. Moves that are beneficial one by one could become counterproductive when they are applied together. Thus, this parallel version leads to a worse likelihood score in some cases, an important problem that does not happen in our version. Additionally, Paramin is meant to be executed in a distributed infrastructure, such as a cluster of computers, and it needs an installed version of the MPI library. On the other hand, our approach focuses on taking advantage of common desktop multicore systems and it only needs to add the appropriate OpenMP compile-time flag to the selected compiler, reducing the computing expertise needed to profit from it.

3. Optimization of the Sequential Program

The profiling of the original application revealed that it could be optimized mainly in three aspects. First, some light class member functions that were invoked very often in the application from other classes could not be inlined by the compiler because they were not expressed in the header files that describe their associated classes, but in the associated implementation files. Moving their definition to the header file allowed their effective inlining, largely reducing their weight in the runtime.

Second, we found several opportunities to avoid recalculations of values, i.e., computations that are performed several times on the same inputs, thus yielding the same result. While some of these opportunities could be exploited just by performing loop-invariant code motion, other more complex situations required a more elaborated approach such as dynamically building tables to cache the results of these calculations and later resorting to these tables to get the associated results.

The third most important optimization applied consisted in modifying the data structure used to represent bidimensional matrices in the heaviest functions. The original code used a layout based on an array of pointers, each pointer allowing the access to a row of the matrix that had been separately allocated. These matrices were changed to be stored using a single consecutive buffer, thus improving the locality, avoiding indirections and helping the compiler to reason better on the code.

4. Parallelization of the Simulated Annealing Algorithm

The SA algorithm used in Gadget is a global optimization method that tries to find the global optimum of a function of N parameters. At the beginning of each iteration, this search algorithm decides which parameter np it is going to be modified to explore the search space. Then, the algorithm generates a trial point with a value of this parameter that is within the step length given by element np of vector $\overrightarrow{VM}$ (of length N) of the user selected starting point by applying a random move. The function is evaluated at this trial point and its result is compared to its value at the initial point. When minimizing a function, any downhill step is accepted and the process repeats from this new point. An

```
1  Algorithm SimulatedAnnealing(p⃗, VM⃗, ns, T, nt, MaxIterations)
       input  : - initial vector of N parameters p⃗ = p_i, 1 ≤ i ≤ N
                - step lengths vector with one element per parameter VM⃗ = VM_i, 1 ≤ i ≤ N
                - frequency of VM⃗ updates ns
                - temperature T
                - frequency of temperature T reductions nt
                - maximum number of evaluations MaxIterations
       output: optimized vector of N parameters p⃗ = p_i, 1 ≤ i ≤ N
2      iterations = 0
3      param_i = i, 1 ≤ i ≤ N                          //permutation of parameters
4      bestLikelihood = evaluate(p⃗)
5      while not terminationTest(bestLikelihood, p⃗) do
6          for a = 1 to nt do
7              reorder(param⃗)
8              for j = 1 to ns do
9                  for i = 1 to N do
10                     iterations = iterations + 1
11                     np = param_i                    //parameter to modify
12                     tmp_p⃗ = p⃗
13                     tmp_p_np = adjustParam(p_np, VM⃗)
14                     likelihood = evaluate(tmp_p⃗)
15                     accept(T, bestLikelihood, likelihood, p⃗, tmp_p⃗)
16                     if iterations = MaxIterations then
17                         return p⃗
18                     end
19                 end
20             end
21             adjustVM(VM⃗)
22         end
23         reduceTemperature(T)
24     end
25     return p⃗
```

Figure 1: Simulated Annealing algorithm

uphill step may be also accepted to escape from local minimum. This decision is made by the Metropolis criteria. It uses a variable called Temperature (T) and the size of the uphill move in a probabilistic manner to decide the acceptance. The bigger T and the size of the uphill move are, the more likely that move will be accepted. If the trial is accepted, the algorithm moves on from that point. If it is rejected, another point is chosen instead for a trial evaluation. Each ns evaluations of the N parameters the elements of $\overrightarrow{VM}$ are adjusted so that half of all function evaluations in each direction will be accepted. Thus, the modification of each element of $\overrightarrow{VM}$ depends on the number of moves accepted along that direction. Also, each nt adjustments to $\overrightarrow{VM}$ the temperature is reduced. Thus, a temperature reduction occurs every $ns \times nt$ cycles of moves along every direction (each $ns \times nt \times N$ evaluations of the function). When T declines, uphill moves are less likely to be accepted and the percentage of rejections rises. Given the scheme for the selection for $\overrightarrow{VM}$, $\overrightarrow{VM}$ falls. Thus, as T declines, $\overrightarrow{VM}$ falls and the algorithm focuses upon the most promising area for optimization. The process stops when no more improvement can be expected or when the maximum number of function evaluations is reached. A pseudocode of the SA algorithm is shown in Figure 1.

This algorithm was parallelized using OpenMP. The followed parallelization pattern consisted in subdividing the search process in a number of steps that are applied in sequence, parallelism being exploited within each step. Since the expensive part of a search algorithm is the evaluation of the fitness function, the parallelism is exploited by distributing the function evaluations across the available threads. The results of the evaluations are stored in shared variables so that all the values can be analyzed by the search algorithm in order to decide whether to finish the search process, and if this is not the case, how to perform the next search step.

This work aims to reduce the execution time without diminishing the quality of the solution. For this purpose, two parallel algorithms were developed: the reproducible and the speculative algorithm. The first one follows exactly the same sequence as the original serial algorithm, and thus it finishes in the same point and with the same likelihood score. The speculative version, however, is allowed to change the parameter modification sequence in order to increase the parallelism, and thus it can finish in a different point and with a different likelihood value. In this latter case, some specification variables of the sequential algorithm were adapted to prevent the algorithm from converging to a worse likelihood value.

In the parallel versions not all the functions evaluations provide useful information to the search process. Thus, the number of `effective` evaluations is taken into account as ending criterion instead of the number of total evaluations.

The same parallelization strategies have been applied to the H&J algorithm, described in Section 5.

4.1. Reproducible version

In this version each thread performs the evaluation of the function with the modification of a different parameter in parallel. For example, with four threads, in the first step, the thread 1 could perform the evaluation changing the first parameter, the thread 2 the second parameter, the thread 3 the third parameter, and the thread 4 the fourth parameter.

From these evaluations the algorithm moves to the first point (in sequential order) that is accepted (directly or applying the Metropolis criteria), dismissing the others. As a result, all the evaluations up to the first one with an accepted point are taken into account to advance in the search process, and for this reason they will be counted as effective evaluations, while all the subsequent simulations are discarded. For the previous example, if the modification to the first parameter is rejected and the modification to the second parameter is accepted, the calculations performed by threads 1 and 2 are considered, and the calculations by threads 3 and 4 are discarded, even if one of them obtains a better likelihood that the obtained by thread 2. In this case, 4 evaluations are performed in parallel but only 2 of them are recorded as effective evaluations.

Following our example, since the last evaluation considered modified the second parameter, in the next step, thread 1 will perform the evaluation modifying the third parameter, thread 2 the fourth parameter, thread 3 the fifth parameter, and thread 4 the sixth parameter. In order to obtain the same result as in the sequential algorithm, the parameters re-evaluated will take the same value as in the previous step. This required changing the way of generating the random values. Namely, while the sequential version relies on `rand`, the reproducible version uses a random function which generates its result from a seed provided by the user. This allows to re-generate previously generated random values by providing the same seed, which is recorded by our implementation for this purpose. Another change was that while the random numbers of the sequential version follow a single sequence, the parallel version uses three different seeds, giving place to three sequences of random numbers. One seed (`seedP`) is used to change the order in which the parameters are modified, another one (`seedM`) is used for the acceptance of the Metropolis criteria, and the last one (`seed`) is used for the calculation of the new value of the parameters.

4.2. Speculative version

As in the reproducible version, each thread performs the evaluation with the modification of a different parameter in parallel, but now the move

with the best likelihood is selected. This point will be accepted if it obtains a better likelihood than the initial point. Otherwise, the Metropolis criteria is applied to decide its acceptance. For example, with 4 threads, in the first step thread 1 could perform the simulation changing the first parameter, thread 2 the second parameter, thread 3 the third parameter and thread 4 the fourth parameter. If thread 3 obtains the best likelihood, the modification of the third parameter will be the one accepted and the other modifications will be discarded. In the next parallel step, thread 1 will work with a change in the fifth parameter, thread 2 in the sixth parameter, thread 3 in the seventh parameter and thread 4 in the eighth parameter.

Note that the speculative version performs and considers evaluations that are never performed in the sequential version. This makes it follow search paths that are different from those of the sequential and the reproducible versions. For this reason, this algorithm can finish in a different point and with a different likelihood value.

In this version, unlike in the reproducible version, some discarded simulations provide information to the search process even though they do not modify the parameters. This forced us to adapt some of the most important variables used during the search process. These variables, which we describe providing their semantics in the sequential version, are:

$\overrightarrow{nacp}$: vector that stores for each parameter the number of times a change in its value has been accepted. It affects the calculation of the step lengths vector $\overrightarrow{VM}$, explained at the beginning of Section 4. Namely, the higher $\overrightarrow{nacp}$, the larger value will have $\overrightarrow{VM}$ and the changes performed in the parameter will be bigger.

ns: scalar that provides the frequency of updates to $\overrightarrow{VM}$.

For this parallel version the above variables were changed as follows:

$\overrightarrow{nacp}$: Its value for a given parameter increases whenever (a) the change evaluated in that parameter improves the likelihood, or (b) the change in that parameter does not improve the likelihood but it happens to be the best one and it is accepted applying the Metropolis criteria. Notice that the second situation implies that none of the changes improved the likelihood.

ns: In the parallel algorithm not all the parameter modifications provide useful information to the search process. Thus, to avoid to change the step lenght associated to each parameter ($\overrightarrow{VM}$ vector) too often, in addition to the global scalar `ns`, a vector $\overrightarrow{vns}$ has been defined with the aim of processing the step length individually. It is increased whenever (a) the change simulated in the parameter improves the likelihood, or (b) the parameter change is rejected (also applying the Metropolis criteria). This way, it is increased in all the situations except when the change would be accepted by the Metropolis criteria.

Also, in order to avoid decreasing the temperature too fast, which would lead to a premature halt of the algorithm, it is necessary to take into account that the number of discarded evaluations increases with the number of threads used in the parallel algorithm. For this reason now each temperature iteration consists of $ns \times numThreads/2$ parallel steps, that is, $ns \times numThreads/2 \times numThreads$ evaluations, where ns is the scalar with the same value as in the sequential version and $numThreads$ is the number of threads used to execute the parallel version.

Finally, the counter of the number of effective evaluations increases every time a parameter change is rejected (also applying the Metropolis criteria) and once for every parallel step.

5. Parallelization of the Hooke & Jeeves Algorithm

Hooke and Jeeves (H&J) [18] is a pattern search method that consists in a sequence of exploratory moves from a base point, followed by pattern moves that provide the next base point to explore.

```
1  Algorithm HookeJeeves(p⃗, delta⃗, MaxIterations, rho)
       input  : - initial vector of N parameters p⃗ = p_i, 1 ≤ i ≤ N
                - modification length for each parameter delta⃗ = delta_i, 1 ≤ i ≤ N
                - maximum number of evaluations MaxIterations
                - reduction control variable rho
       output: optimized vector of N parameters p⃗ = p_i, 1 ≤ i ≤ N
2      iterations = 0
3      tmp_p⃗ = p⃗
4      bestLikelihood = evaluate(p⃗)
5      initialLikelihood = bestLikelihood
6      while not terminationTest(iterations, MaxIterations, bestLikelihood, p⃗) do
7          iterations = iterations + 1
8          for i = 1 to N do
9              tmp_p_i = p_i + delta_i
10             likelihood = evaluate(tmp_p⃗)
11             if likelihood < bestLikelihood then
12                 bestLikelihood = likelihood
13             else
14                 tmp_p_i = p_i − delta_i
15                 likelihood = evaluate(tmp_p⃗)
16                 if likelihood < bestLikelihood then
17                     bestLikelihood = likelihood
18                 else
19                     tmp_p_i = p_i
20                 end
21             end
22         end
23         adjust(p⃗, tmp_p⃗, delta⃗, rho, initialLikelihood, bestLikelihood)        //includes exploratory moves
24         p⃗ = tmp_p⃗
25     end
26     return p⃗
```

Figure 2: Hooke & Jeeves algorithm

The exploratory stage performs local searches in each direction by changing a single parameter of the base point in each move. For each parameter, the algorithm considers first an increment *delta* in the positive direction. If the function value in this point is better than the old one, then the algorithm selects this new point like the new base point. Otherwise, an increase *delta* is done in the negative direction, and if the result is better than the original one, then the algorithm selects this new point as the base point. Otherwise the algorithm keeps the initial value and proceeds to evaluate changes in the next parameter. Once all the parameters have been explored, the algorithm continues with the pattern moves stage.

In the pattern stage, each one of the parameters is increased by an amount equal to the difference between the present parameter value and the previous one (its value before the exploratory stage). The aim is to move the base point towards the direction that improved the result during the previous stage. Then, the function is evaluated in this new point. If the function value is better, this point becomes the new base point for the next exploratory moves. Otherwise, the pattern moves are ignored.

The search proceeds in series of these two stages until a minimum is found or the maximum number of evaluations is reached. If after a exploratory stage the base point does not change, *delta* is reduced. The amount of the reduction is determined by a user-

supplied parameter called *rho*. Taking big steps gets to the minimum more quickly, at the risk of stepping right over an excellent point. Figure 2 summarizes this search algorithm.

The parallel implementations of this algorithm use nested parallelism to parallelize the exploratory stage. At the top level, each thread is in charge of evaluating the changes on a specific parameter. Then, each thread is split in two threads, so that one of them evaluates the movement in the positive direction and the other one the movement in the negative direction. For this reason, the required number of threads to execute these versions is a multiple of 2. For example, if we have 4 threads, in the first parallel evaluation one thread will evaluate parameter 1 + *delta*, another one the parameter 1 - *delta*, another one the parameter 2 + *delta* and finally another one the parameter 2 - *delta*.

5.1. Reproducible version

The reproducible version follows exactly the same sequence as the original sequential algorithm. This version chooses the first move (in sequential order) that improves the likelihood as base point for the next search step. All the subsequent evaluations are disregarded and they will not be taken into account to increase the counter of the number of effective evaluations.

For the previous example using 4 threads, if the first evaluation that obtains a better likelihood corresponds to the movement of the first parameter in the negative direction (parameter 1 - *delta*), the two evaluations using the second parameter are discarded (parameter 2 + *delta* and parameter 2-*delta*). The next step will start from parameter 2 and it will evaluate the moves in the positive and negative direction of both parameters 2 and 3.

5.2. Speculative version

This version chooses the move that gives place to the best likelihood as base point for the next search step. Since all the evaluations are taken into account to choose this value, if n was the last parameter considered in a parallel step, this version always starts the evaluations of the next parallel step from parameter $n + 1$.

For this version the counter of the number of effective evaluations is only increased in two situations. First, for every parameter in which none of the movements improved the likelihood, the counter is increased by two units, to account for the two movements tested. Second, among all those movements that improve the likelihood, only the best one in the parallel step is considered. If this movement is in the positive direction, the counter is increased by one unit, otherwise it is increased by two units, because negative movements are always evaluated after a positive movement has been discarded.

Finally, the outer loop that iterates on the parameters to perform the exploratory moves has step $numThreads/2$, since each parallel step evaluates the two possible movements for $numThreads/2$ parameters.

6. Experimental Results

The different parallel implementations were evaluated in an HP ProLiant XL230a Gen9 server running the Scientific Linux release 6.4. It consists of two processors Intel Haswell E5-2680 at 2.5 GHZ with 12 cores per processor and 64 GB of memory. All the codes were compiled with the gnu g++ compiler version 4.9.1 and the compilation flag -O3. Three different models were used to better evaluate the results:

IEO: This model is used by the IEO to assess the southern hake stock. It counts with 63 parameters to optimize.

TUSK: It is a single-species model of tusk (*brosme brosme*) in Icelandic waters which is used by ICES as the basis for catch advice. It counts with 48 parameters to optimize.

HADOCK: It is a single-species, single-area model used to model the Icelandic haddock. It is the

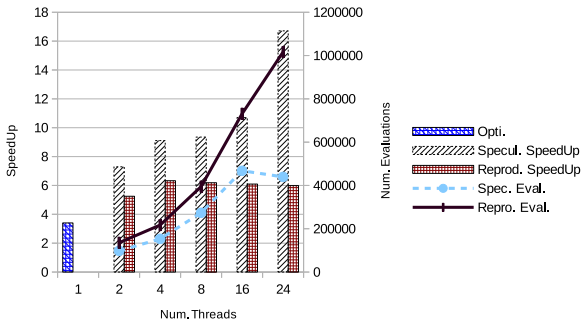
	SA	H&J	SA + H&J
IEO	5264	5266	10504
TUSK	4699	4741	9372
HADOCK	1178	1177	2362

Table 1: Execution times, in seconds, of the original code using a single search algorithm or both

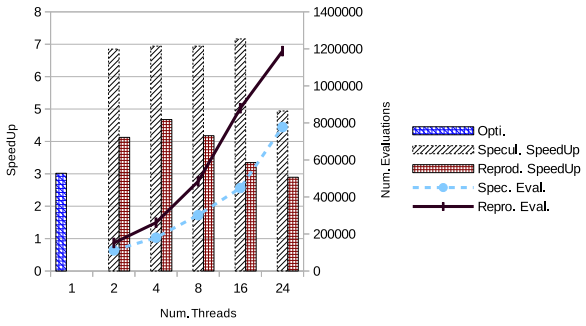
example model provided with Gadget. It is available for download from the Gadget website. It counts with 38 parameters to optimize.

The performance of the parallel algorithms was evaluated in terms of both the computational efficiency (speedup) and the quality of the solutions (likelihood score obtained). All the speedups were calculated with respect to the execution time of the original sequential application (Gadget v2.2.00, available at www.hafro.is/gadget/files/gadget2.2.00.tar.gz). These execution times are shown in Table 1, and they correspond to the whole execution of the application, either using a single search algorithm or two. In all the experiments the optimization algorithms stop when the maximum number of effective evaluations (1000 for both algorithms) is reached.

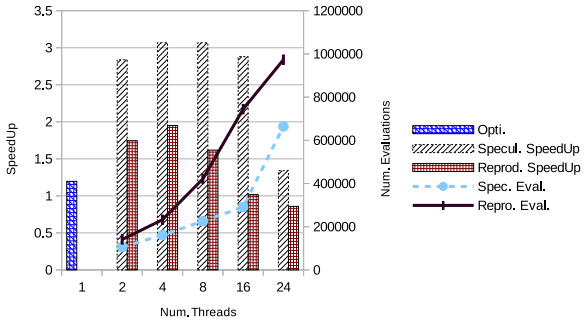
Figure 3 shows the speedups for the optimized sequential version and the two parallel versions developed, the reproducible one and the speculative one, when using only the SA algorithm, for the 3 model examples considered and for different number of threads. The total number of evaluations performed for the parallel algorithms is also shown in the figures. The optimization of the sequential code obtains a significant reduction in the execution time, achieving a speedup of 3.3 for the IEO model. As regards the parallel versions, the number of performed evaluations increases with the number of threads, being this increment larger for the reproducible version, as a greater number of evaluations has to be discarded in this case. For this reason, the speedup of the reproducible version does not improve beyond 4 cores. The speculative version behaves better, although the results depend significantly on the example model. The best results are for the IEO model because in this case



(a) IEO



(b) TUSK

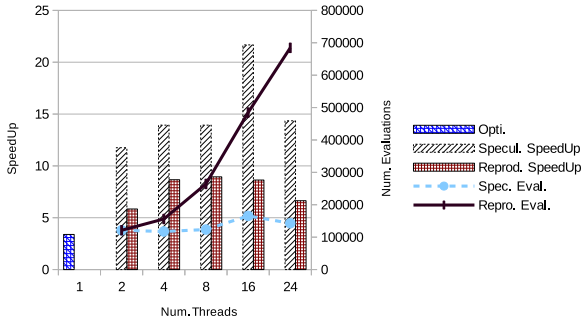


(c) HADOCK

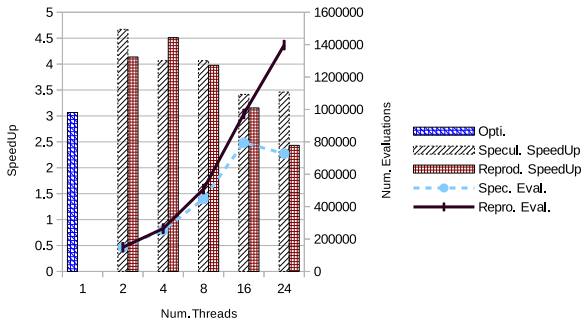
Figure 3: Speedups using the SA algorithm, fill bars indicate relative speedup and the lines the number of function calls as a function of number of threads and by approach.

the starting point is closer to the optimum point (see Table 2, discussed at the end of this section), which increases the rate of rejects and thus, the number of

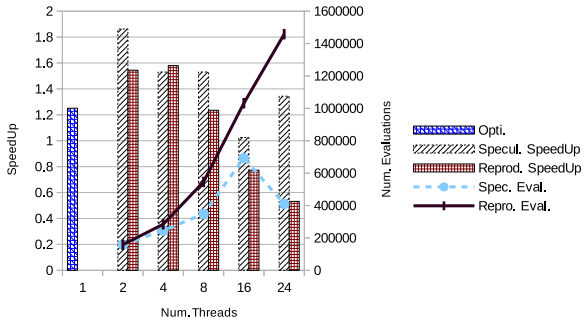
effective evaluations. Note that for this example the number of total evaluations does not increase as fast as in the other models.



(a) IEO

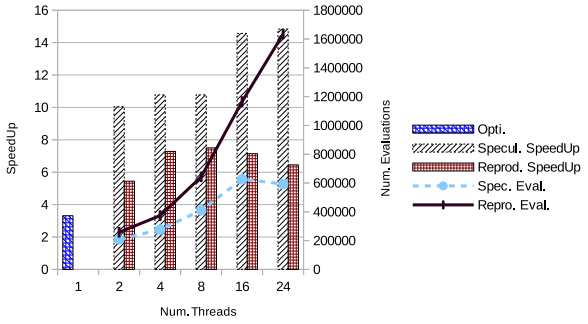


(b) TUSK

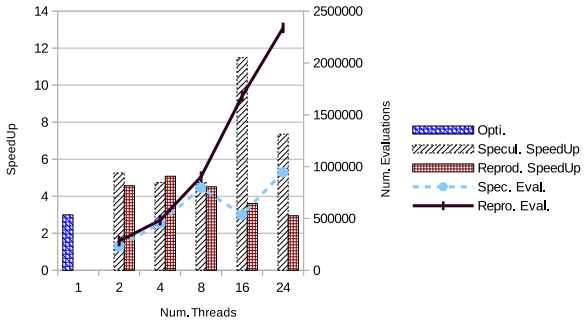


(c) HADOCK

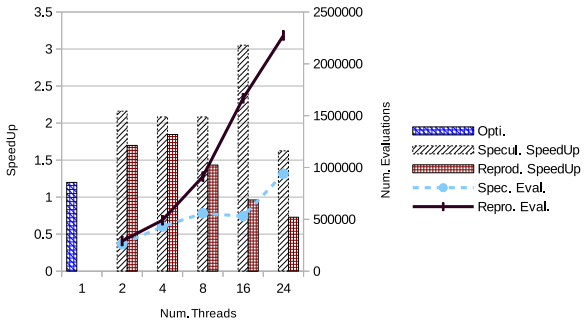
Figure 4: Speedups using the H&J algorithm, fill bars indicate relative speedup and the lines the number of function calls as a function of number of threads and by approach.



(a) IEO



(b) TUSK



(c) HADOCK

Figure 5: Speedups using the SA and the H&J algorithms, fill bars indicate relative speedup and the lines the number of function calls as a function of number of threads and by approach.

Similar results are obtained for the H&J algorithm, as can be seen in Figure 4. Finally, Figure 5 shows

	IEO	TUSK	HADOCK
Initial	1015.1691	26128.3060	0.96417375
Sequential	1015.1130	6537.8195	0.85868597
2	1015.1120	6539.2886	0.86255539
4	1014.8995	6511.2774	0.85186922
8	1015.0495	6509.1936	0.85107118
16	1015.0250	6507.5369	0.85250038
24	1015.0400	6507.1087	0.85083064

Table 2: Likelihood obtained using SA

	IEO	TUSK	HADOCK
Initial	1015.1691	26128.3060	0.96417375
Sequential	1015.0780	6837.3983	0.87042278
2	1015.0780	6837.3983	0.87042278
4	1015.1062	6953.9042	0.85509464
8	1015.1045	6891.2857	0.86385086
16	1015.1081	6898.9408	0.86559744
24	1015.0817	6963.9621	0.86008887

Table 3: Likelihood obtained using H&J

the speedups when using the SA and the H&J algorithms in sequence. This hybrid algorithm combines the power and robustness of the SA algorithm with the faster convergence of a local method as the H&J algorithm. It is a common choice as, in general, it is an effective approach to find an optimum point. Note that the speedup of the reproducible version only improves up to 4 cores, whereas the speculative version obtains the best results when using 16 or 24 cores. The best results are for the IEO model, where the execution time is reduced from 175 to 23 minutes using the reproducible version and 4 cores and to only 12 minutes using the speculative version and 24 cores.

As regards the quality of the solution, the reproducible versions obtain exactly the same likelihood score as the original sequential version. The scores for the speculative version for SA, H&J and the hybrid SA-H&J algorithms are shown in Tables 2, 3 and 4, respectively. The tables also show for comparative purposes the starting likelihood value associated to the input (Initial row in the tables) and the value after applying the sequential optimization algorithms (Sequential row). Notice that the likelihood value is better the lower it is, thus the search algorithms reduce it. We can see that some models, like IEO, begin with values near the optimum point found, while others, such as TUSK, can be strongly optimized by Gadget. As expected, the best likelihood value is obtained with the hybrid algorithm and this value is even improved when using the parallel speculative version.

	IEO	TUSK	HADOCK
Initial	1015.1691	26128.3060	0.96417375
Sequential	1015.0478	6511.6363	0.85396624
2	1015.0641	6515.1060	0.85601157
4	1014.8474	6508.4827	0.85074075
8	1014.9955	6507.3761	0.85070340
16	1014.9662	6507.0652	0.85150650
24	1014.9820	6507.0435	0.85058013

Table 4: Likelihood obtained using SA & H&J

7. Conclusions

The aim of this work has been to speedup the Gadget program to reach a reliable model in a reasonable execution time, getting profit from the new multicore architectures. First, the sequential code was analyzed and optimized so that the most important bottlenecks were identified and reduced. Then, the SA and H&J algorithms, used to optimize the model provided by Gadget, were parallelized using OpenMP. Two different versions were implemented for each algorithm, the speculative one, which yields the same result as the sequential version, and the speculative version, which can exploit more parallelism. It must be stressed that all the parallel algorithms proposed are totally general and can thus be applied to other optimization problems. As expected, the speculative version provides better results for all the analyzed examples, achieving a speedup of 14.8 (4.5 with respect to the optimized sequential version) for the hybrid SA-H&J algorithm and the IEO model on a 24-core server. Moreover, the speculative version not only reduces significantly the execution time, but it also

obtains a better likelihood score.

OpenMP is nowadays the standard de facto for shared memory parallel programming and it allows the efficient use of today's mainstream multicore processors. The OpenMP versions of the Gadget software developed in this work will allow researchers to make a better use of their computer resources. They are publicly available under GPLv2 license at `www.github.com/hafro/gadget`.

Acknowledgment

This work has been partially funded by the Galician Government (Xunta de Galicia) and FEDER funds of the EU under the Consolidation Program of Competitive Research Units (Network ref. R2014/041) and by European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no.613571. Authors gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the HP ProLiant XL230a Gen9 server and Santiago Cerviño from IEO for his support.

References

[1] J. J. Pella, P. K. Tomlinson, A generalized stock production model, Inter-American Tropical Tuna Commission, 1969.

[2] K. Patterson, R. Cook, C. Darby, S. Gavaris, L. Kell, P. Lewy, B. Mesnil, A. Punt, V. Restrepo, D. W. Skagen, et al., Estimating uncertainty in fish stock assessment and forecasting, Fish and Fisheries 2 (2) (2001) 125–157.

[3] G. Einarsson, Competitive coevolution in problem design and metaheuristical parameter tuning, M.Sc. Thesis. University of Iceland, Iceland (2014)`http://hdl.handle.net/1946/18534`.

[4] A. Punt, B. Elvarsson, Improving the performance of the algorithm for conditioning implementation simulation trials, with application to north atlantic fin whales, IWC Document SC/D11/NPM1 (7pp).

[5] J. Begley, D. Howell, An overview of Gadget, the globally applicable area-disaggregated general ecosystem toolbox, ICES, 2004.

[6] J. Begley, Gadget user guide, `http://www.hafro.is/gadget/files/userguide.pdf` (2012).

[7] L. Taylor, J. Begley, V. Kupca, G. Stefansson, A simple implementation of the statistical modelling framework Gadget for cod in Icelandic waters, African Journal of Marine Science 29 (2) (2007) 223–245.

[8] H. Björnsson, T. Sigurdsson, Assessment of golden redfish (sebastes marinus l) in icelandic waters, Scientia Marina 67 (S1) (2003) 301–314.

[9] B. Elvarsson, L. Taylor, V. Trenkel, V. Kupca, G. Stefansson, A bootstrap method for estimating bias and variance in statistical fisheries modelling frameworks using highly disparate datasets, African Journal of Marine Science 36 (1) (2014) 99–110.

[10] E. Andonegi, J. A. Fernandes, I. Quincoces, X. Irigoien, A. Uriarte, A. Pérez, D. Howell, G. Stefánsson, The potential use of a Gadget model to predict stock responses to climate change in combination with bayesian networks: the case of Bay of Biscay anchovy, ICES Journal of Marine Science: Journal du Conseil 68 (6) (2011) 1257–1269.

[11] S. McCully, F. Scott, L. Kell, J. Ellis, D. Howell, A novel application of the Gadget operating model to North East Atlantic porbeagle, Collect. Vol. Sci. Pap. ICCAT 65 (6) (2010) 2069–2076.

[12] D. Howell, B. Bogstad, A combined Gadget/FLR model for management strategy evaluations of the Barents Sea fisheries, ICES Journal of Marine Science: Journal du Conseil 67 (9) (2010) 1998–2004.

[13] Report of the working group on the assessment of southern shelf stocks of hake, monk and megrim, Tech. Rep. ICES CM 2010/ACOM:11,

International Council for the Exploration of the Sea (2010).

[14] Report of the benchmark workshop on deep-sea stocks (wkdeep), Tech. Rep. ICES CM 2014/ACOM:44, International Council for the Exploration of the Sea.

[15] Report of the benchmark workshop on redfish management plan evaluation (wkredmp), Tech. Rep. ICES CM 2014/ACOM:52, International Council for the Exploration of the Sea.

[16] Report of the inter benchmark process on greenland halibut in ices areas i and ii (ibphali), Tech. Rep. ICES CM 2015/ACOM:54, International Council for the Exploration of the Sea.

[17] A. Corana, M. Marchesi, C. Martini, S. Ridella, Minimizing multimodal functions of continuous variables with the 'Simulated Annealing' algorithm, ACM Transactions on Mathematical Software (TOMS) 13 (3) (1987) 262–280.

[18] R. Hooke, T. A. Jeeves, Direct search solution of numerical and statistical problems, Journal of the ACM (JACM) 8 (2) (1961) 212–229.

[19] D. P. Bertsekas, Nonlinear programming, Athena scientific, 1999.

[20] OpenMP website, `http://openmp.org/`.

[21] D. R. Greening, Parallel simulated annealing techniques, Physica D: Nonlinear Phenomena 42 (1) (1990) 293–306.

[22] E. E. Witte, R. D. Chamberlain, M. Franklin, et al., Parallel simulated annealing using speculative computation, Parallel and Distributed Systems, IEEE Transactions on 2 (4) (1991) 483–494.

[23] D. J. Ram, T. Sreenivas, K. G. Subramaniam, Parallel simulated annealing algorithms, Journal of parallel and distributed computing 37 (2) (1996) 207–212.

[24] E. Onbaşoğlu, L. Özdamar, Parallel simulated annealing algorithms in global optimization, Journal of Global Optimization 19 (1) (2001) 27–50.

[25] D.-J. Chen, C.-Y. Lee, C.-H. Park, P. Mendes, Parallelizing simulated annealing algorithms based on high-performance computer, Journal of Global Optimization 39 (2) (2007) 261–289.

[26] A. Bevilacqua, A methodological approach to parallel simulated annealing on an smp system, Journal of Parallel and Distributed Computing 62 (10) (2002) 1548–1570.

[27] D. R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.

[28] G. Stefánsson, A. Jakobsdóttir, B. Elvarsson, J. Begley, Paramin, a composite parallel algorithm for minimising computationally expensive functions, `http://www.hafro.is/gadget/files/paramin.pdf` (2004).

[29] A. Geist, PVM: Parallel Virtual Machine: a users' guide and tutorial for networked parallel computing, MIT press, 1994.

[30] MPI Forum, Message Passing Interface, `http://www.mpi-forum.org`.