Architecture, implementation and parallelization of the software to search for periodic gravitational wave signals

G.Poghosyan[a], S.Matta[a,b], A.Streit[a], M.Bejger[c], A.Krolak[d]

[a]*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany*
[b]*PEC University of Technology, 160012 Chandigarh, India*
[c]*N. Copernicus Astronomical Center of the Polish Academy of Sciences, 00-716 Warsaw, Poland*
[d]*Institute of Mathematics of the Polish Academy of Sciences, 00-956 Warsaw, Poland*

Abstract

The parallelization, design and scalability of the `PolGrawAllSky` code to search for periodic gravitational waves from rotating neutron stars is discussed. The code is based on an efficient implementation of the $\mathcal{F}$-statistic using the Fast Fourier Transform algorithm. To perform such an analysis of data from the advanced gravitational wave detectors LIGO and Virgo that will start operating at the end of 2015, hundreds of millions of CPU hours will be required - the implementation on high performance computers is therefore mandatory. We have parallelized parts of code using the Message Passing Interface library, implemented a task-farming mechanism for combining the searches at different sky-positions and frequency bands into one extremely scalable program. We have used the MPI I/O library to escape bottlenecks, when writing the generated data into file system. This allowed to develop a highly scalable computation code, which would enable the data analysis at large scales on acceptable time scales. Benchmarking of the code on a Cray XE6 system was performed to show efficiency of our parallelization concept and to demonstrate scaling up to 50 thousand cores in parallel.

Keywords: Parallelization, MPI, MPI I/O, HPC, gravitational waves, $\mathcal{F}$-statistic, multi-level parallelism, Farm skeletons

1. Introduction

Gravitational waves (GWs) - variations of the curvature of spacetime, able to propagate through spacetime in a wave-like fashion - were first predicted by Albert Einstein (Einstein, 1916), and they are a direct consequence of the general theory of relativity that he proposed. Several properties of GWs are similar to those of electromagnetic waves. GWs also propagate with speed of light and are polarized (two polarizations in the description of general relativity). The best empirical, yet *indirect* evidence for gravitational radiation comes form the observations of tight relativistic binary pulsar systems; first such a system was discovered by R. Hulse and J. Taylor with the radio observations from the Arecibo telescope (Hulse & Taylor, 1975). Direct detection of GWs will constitute a very precise test of Einstein's theory of relativity and open a new field - GW astronomy. Currently, the most promising GW detector concept is of the Michelson-Morley interferometer type. The detection principle is as follows: while a GW passes through such a detector, it changes the length of its arms and affects the interference pattern of the laser light circulating in the interferometer (Saulson, 1994).

State-of-the-art interferometric GW detectors, LIGO[1] in the USA and Europe (Italian-French, with the contribution of Hungary, the Netherlands and Poland) Virgo[2] have collected a large amount of data that are still being analyzed. Meanwhile, the advanced LIGO and Virgo detectors are under construction and they are forecasted to start collecting new, more sensitive data at the end of 2015. It is expected that these advanced detectors will be sufficiently sensitive so that the direct detection of GWs can finally be achieved. As the GW signals are extremely weak, their detection constitutes a major challenge in data analysis and computing. Several types of astrophysical GW sources are investigated: coalescence of compact binaries containing neutron stars and black holes, supernova explosions, quantum effects in the early Universe as well as rotating, non-axisymmetric neutron stars.

The departure from axisymmetry in the mass distribution of a rotating neutron star can be caused by strong magnetic fields and/or elastic stresses in its interior. The search for such long-lived, periodic GW signals generated by the spinning star is nevertheless particularly computationally intensive. This is because the GW signal is very weak and one needs to analyze long stretches of data in order to extract the signal "buried" in the noise of the detector. Due to this, the modulation of the signal due to the motion of the detector with respect to the solar system barycenter has to be taken into account; it depends on the location of the source and a modulation that is a function of the intrinsic change of rotation frequency of the deformed neutron star. Moreover, we do not know the polarization, amplitude, and phase of the GW signal. Consequently, the parameter space to search for the signal becomes very large.

The *Polgraw-Virgo* team, working within the LIGO scientific

[1]http://www.ligo.org
[2]https://wwwcascina.virgo.infn.it

Collaboration (LSC) and the Virgo Collaboration has developed algorithms and a pipeline called `PolGrawAllSky` to search for GW signals from spinning neutron stars (Astone et al., 2010). The pipeline was applied to the analysis of the data gathered by the Virgo detector during its first science run denoted as VSR1. The analysis involved 5 million CPU hours and took almost three years to complete (Aasi et al., 2014). The serial code's design allowed to use only one processor core and was run on a number of computer clusters with standard queuing systems. This performance turned out to be not entirely satisfactory for current and future requirements of the GW data analysis. To analyze all the data collected by the Virgo detector, 250 million CPU hours are required whereas analysis of all the data that will be collected from the advanced detectors expected to be available by the year 2018 will require four times more resources, i.e., 1000 million CPU hours. To perform this analysis one would need 1petaFLOPS computer working continuously for one year.

To estimate the computational requirements we have performed representative tests with the Gaussian noise data at different band frequencies illustrated on Fig. 1 and 2. For example, a serial search for GWs at frequencies 600, 1000, 1700 and 2000 will require a total of 20 thousand CPU hours of computation, which is more than two years on a single CPU and correspondingly the output generated by this simulation would be ca. 4GB.

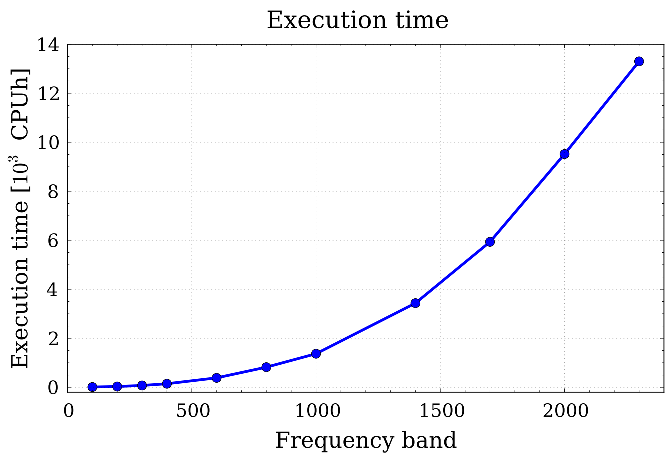


Figure 1: Total execution time of embedded `PolGrawAllSky` code in thousand CPU hours as a function of frequency band.

To alleviate the computations, this paper proposes a massively parallelized version of the `PolGrawAllSky` code that uses the Message Passing Interface (MPI) library (Message Passing Interface Forum, 2012). MPI is a distributed memory parallelization scheme commonly used in high performance computing (HPC). This parallelized version is able to run on high performance computers with tens of thousands of cores. We are reporting a sufficient performance increase of the parallel `PolGrawAllSky` code enabling its usage on massively parallel HPC systems for production analysis of data already collected by GW detectors and also of data from the advanced GW detectors that will start to be available by the end of the year 2015 (Aasi et al., 2013b).

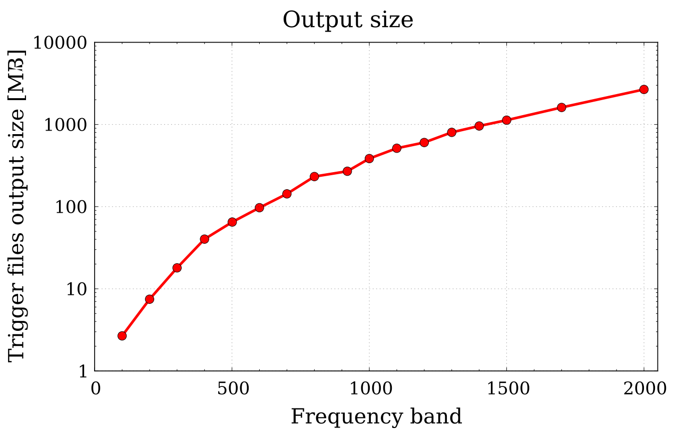


Figure 2: Total output of `PolGrawAllSky` code in Megabytes as a function of frequency band.

1.1. Mathematical Methodology

The algorithms to search for gravitational wave signals from rotating neutron stars implement the $\mathcal{F}$-statistic (Jaranowski et al., 1998), derived by one of us and commonly adopted in other pipelines (see e.g., Aasi et al. 2013a). By using the $\mathcal{F}$-statistic one doesn't need to search for the polarization, amplitude and phase of the signal. Instead, one is left with a 4-dimensional space parameterized by the GW frequency, frequency derivative (spindown, reflecting the fact that the pulsar is spinning down) and the two angles determining the location of the source in the sky. To implement a computationally efficient algorithm we are faced with two problems. Firstly one would like to minimize the number of grid points on which the $\mathcal{F}$-statistic is evaluated, achieving at the same time a certain target sensitivity of the search. This is equivalent to a well-known geometrical problem called the *covering problem*. Secondly, we would like to take advantage of the Fast Fourier Transform (FFT) algorithm. The FFT algorithm cannot be directly implemented in the calculation of the $\mathcal{F}$-statistic because of the modulation of the signal due to the detector's movement around the Sun. In order to implement it one needs to interpolate the data (so called re-sampling). However, the re-sampling means an additional computational cost that can offset the advantage of the FFT algorithm. Moreover, the FFT algorithm evaluates the $\mathcal{F}$-statistic at some specific values of the frequency called the Fourier frequencies. These frequencies need to be reconciled with the grid points obtained by the solution of the covering problem. Astone et al. (2010) describes in detail how the covering problem and efficient usage of the FFT can be achieved: by constraining the solution it is ensured that one needs to perform the computationally intensive re-sampling procedure only once per sky position. Each sky position corresponds to a grid of values of frequency derivative - the re-sampling occupies therefore only a fraction of the total computational time. Thus we have an algorithm that involves both the evaluation of the $\mathcal{F}$-statistic at smallest number of points and takes the advantage of the FFT at the same time.

The main computation consists of 3 loops - two external loops over the sky positions, an inner loop over the frequency derivatives (spindowns) and finally the FFT execution that evaluates the $\mathcal{F}$-statistic on a grid of frequencies.

2. Description of the code for searching of GW signals

The data time series from a detector is divided into two-day time slots, numbered by an integer d, that we call *frames*. A typical number of time frames in a science run is around one hundred. Each frame of data is divided into narrow frequency bands of 1 Hz width each. The bands overlap by 2^{-5} Hz and they are numbered by integer b. The relation between the frequency $f_{\rm off}$ of the lower edge of the band and the band number b is given by

$$f_{\rm off} = 100 + (1 - 2^{-5})b. \tag{1}$$

Given that the interferometric detectors span the frequency range between $\sim$ 10 and $\sim$ 1000 Hz, in a typical search the number of bands is around one thousand. Thus one has about *one hundred thousand* of time-frequency data sets to analyze.

The PolGrawAllSky code analyzes a given band b in a given time frame d. The input data consist of three files: narrowband time series of the detector data xdat_d_b.bin, ephemeris of the detector DetSSB.bin and grid generating matrix grid.bin. The data xdat_d_b.bin spans the length of 2 sidereal days and is sampled at 0.5 s, thus consisting of $N = 344656$ double precision numbers.

Detector ephemeris DetSSB.bin file contains the 3-dimensional vector, relating the detector to the Solar System Barycenter (SSB), of the same length as the time series data, as well as two additional parameters: phase ϕ_o determining the position of the detector at the time of the first sample of the xdat_d_b.bin file, and ϵ, which is the obliquity of the ecliptic. Thus DetSSB.bin contains $3 \times N + 2 = 1033970$ double precision numbers.

The file grid.bin contains a $4{\times}4$ grid lattice generating matrix M. The rows of the matrix M are four vectors spanning our 4-dimensional parameter space. The integer multiples of these vectors define the grid points in the parameter space where the $\mathcal{F}$-statistic is calculated. As already mentioned in Sect. 1.1, the code performs 3 loops - two external ones over the sky positions, and an inner loop over the range of frequency derivatives (spindowns). The sky positions are transformed from the usual astronomical equatorial coordinates into two integers $\mathbf{n}$ and $\mathbf{m}$ and every spindown value is transformed to an integer denoted by $\mathbf{s}$.

The flow diagram of the main PolGrawAllSky code, underlying its most important features, is presented in Fig. 3; a detailed description can be found in Astone et al. (2010). The first step is to read the data and store it in the memory. Then we start the two loops over the sky, i.e., the range of integers $\mathbf{n}, \mathbf{m}$ described by the grid. First the sky position in equatorial coordinates is recovered. Then the two amplitude modulations and the phase modulation are calculated and the signal is demodulated. Then we perform resampling (interpolations) using the FFT and splines. The search grid is constructed in such a way that we need to perform resampling only once for each sky position, thus allowing for re-using the same resampled and demodulated data in the inner spindown loop. For each $\mathbf{s}$ value in the inner spindown loop we perform the demodulation. This is followed by two Fourier transforms, each for one amplitude demodulation. We then perform interpolation of the FFTs resulting in Fourier transform twice as long as the original one. This interpolation is applied in order to prevent excessive loss of the signal-to-noise ratio as the parameters of the true signal do not necessarily coincide with parameters of the grid points (Astone et al. 2010, Sect. VIB). Finally, for each $\mathbf{s}$ value the $\mathcal{F}$-statistic is calculated. Whenever the value of the $\mathcal{F}$-statistic crosses a predetermined threshold $\mathcal{F}_0$, we register the parameters of this grid point (sky position, frequency and spindown), together with the value of the $\mathcal{F}$-statistic. This set of 5 double precision numbers constitutes the *candidate signal* output. The candidate signals obtained from the analysis of xdat_d_b.bin data are stored in the file named candidates_d_b.bin. The candidate files are then subject of analysis by post-processing codes to extract true gravitational wave signals (if no statistically significant gravitational wave signal is found, an upper limits for the amplitude of the gravitational wave at a given frequency can be obtained).

3. Parallelization of sky loops

As the sky positions are independent of each other, this feature can be exploited to parallelize the outer (sky) loop (see Fig. 3). Current parallel version keeps the inner (spindown) loop over the frequency derivatives. We recall that the inner loop reuses the demodulated and re-sampled data, as described in Astone et al. (2010). Our choice of the parameter space is such that the number of frequency spindowns is a linear function of the band frequency f, whereas the total number of sky positions $N_{\rm sky}$ is a quadratic function of the band frequency. It is evident that the relation $N_{\rm sky}(f)$ plays a crucial role in the parallelization strategy of our computations.

Computation of spindowns for each sky position is distributed onto the available parallel tasks using the MPI point-to-point and collective communication routines implemented in it. Each parallel task makes $P_{\rm size}$ sky positions computations, where $P_{\rm size} = \lceil N_{\rm sky}/N_{\rm tasks} \rceil$ is the ratio of the number of sky positions to the number of parallel tasks, rounded up to the nearest integer. To reach optimal load balancing, computations are distributed to parallel tasks (see Fig. 4) using the round-robin (RR) scheduling algorithm (Kleinrock, 1964).

The tests of scaling performance of the parallel PolGrawAllSky code on computer clusters with up to 1000 CPU cores were encouraging, but not entirely satisfactory. Such a facility is nowadays available in many universities or research computing centers. However, one needs to compute continuously without maintenance for several years in order to analyze a significant fraction of data collected by the LIGO and Virgo detectors on small size supercomputers. Hence, the only solution is to increase the number of parallel tasks by at least

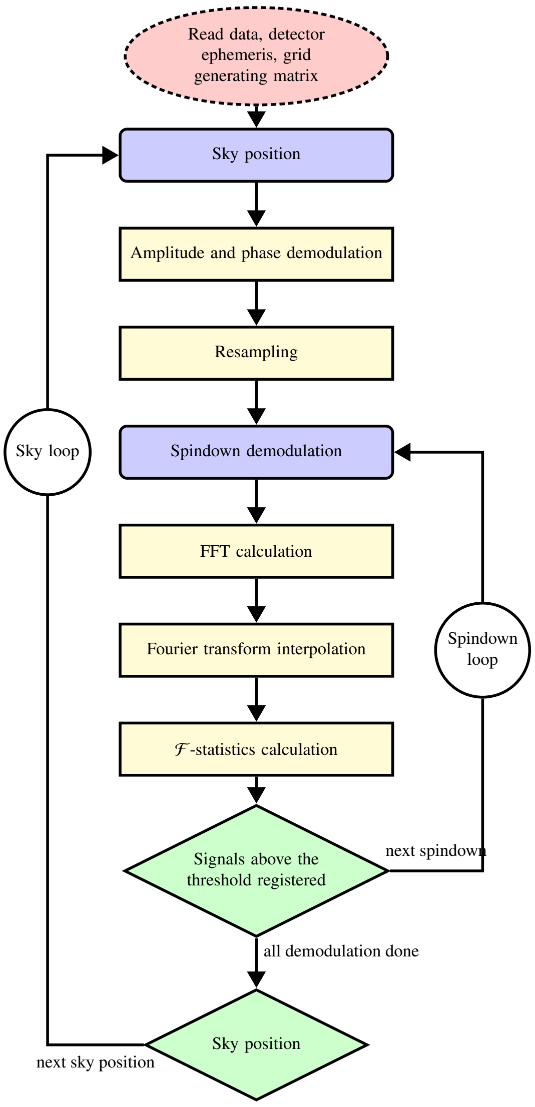

Read data, detector ephemeris, grid generating matrix
Sky position
Amplitude and phase demodulation
Resampling
Spindown demodulation
FFT calculation
Fourier transform interpolation
$\mathcal{F}$-statistics calculation
Signals above the threshold registered
next spindown
Spindown loop
Sky loop
all demodulation done
Sky position
next sky position

Figure 3: Flow diagram of the core code for searching GW signals.

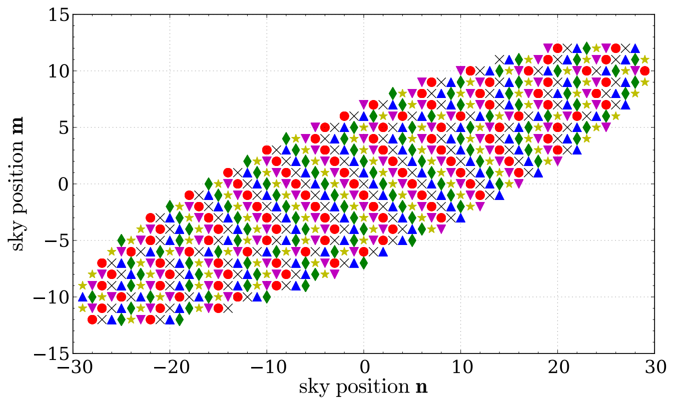


Figure 4: Distribution of computations of the sky positions for frequency band $b = 100$ and for $N_{tasks} = 6$. Six parallel tasks are represented by different symbols and are repeatedly covering the whole hemisphere based on the `round-robin` scheduling algorithm.

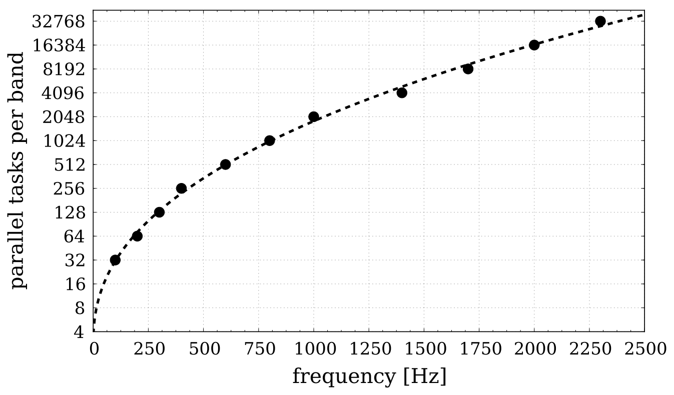


Figure 5: Optimal amount of parallel tasks for a given frequency band. The number of parallel tasks given on the y-axis is required in order to analyze the corresponding frequency band on the x-axis within an hour.

an order of magnitude to be able to speak about acceptable execution times for the full analysis of the data.

Using a limited amount of available computational resources provided within the framework of a PRACE project[3] we have tested the scalability of the parallelized `PolGrawAllSky` code for frequencies up to 2300 Hz, using up to 32768 parallel tasks. The PRACE grant was also useful to estimate the optimal number of parallel tasks per frequency band, when the time necessary to complete the analysis of any band was restricted (an example for one hour restriction time is shown in Fig. 5).

Estimation of the optimal amount of parallel tasks per frequency band allowed us to combine a set of parallel computations of sky loops into one, in order to perform the computation on massively parallel processing (`MPP`) systems. We have used the fact that not only the sky positions, but also searches for GW signal candidates at different frequency bands are independent of each other. Such a low coupled system allows the use of distributed network computing systems such as volunteer systems or grid/cloud systems. Contrary to distributed systems however, executing parallel tasks on `MPP` systems make scheduling and work-flow problems easier to avoid, e.g., by using built-in schedulers and process manager mechanisms specific for the `MPP` systems (Gropp & Lusk, 1995).

3.1. Algorithmic outline of `skyfarmer`

To enhance the scalability of execution of many computations in parallel, we combine many instances consisting of different `PolGrawAllSky` executions that use different numbers of parallel sub-tasks. This feature is implemented using the dynamic process creation and grouping framework of MPI, with different MPI sub-worlds also known as virtual groups, that enables collective and dynamic communication operations across a subset of related tasks. The main `PolGrawAllSky` code with parallel sky loop is encapsulated into another code, named `skyfarmer` , equipped with internal scheduling and bookkeeping mechanism. The `skyfarmer` flow chart is depicted in Fig. 6. For each sky search a domain group and associated communicator as a new virtual MPI sub-world is created,

[3]PRACE Preparatory Access type B 2010PA1183 for 50 000 core-hours on GCS@HLRS, Germany

which allows to execute instances of PolGrawAllSky without global source re-engineering. This concept, known as algorithmic skeleton (AS), was proposed for parallel programming to simplify not only the programming but also to enhance the portability of parallel applications by abstracting from the underlying hardware Cole (1989, 2004). We have developed our own version of farm skeleton skyfarmer that is optimized for running the PolGrawAllSky and is able to reach higher scalability on multi-node system using MPI, as compared to other AS tools Ernsting & Kuchen (2012); Gonzalez-Velez & Leyton (2010). The new communicators are simply created and managed by skyfarmer transparently from the point of view of the embedded parallel PolGrawAllSky , whose tasks run as if in a stand-alone mode with a uniquely assigned communicator.

The structure of the skyfarmer is divided into five main parts:

1. initialization and estimation of the available and necessary parallel resources,
2. construction of different tasks as groups,
3. distribution and decomposition of groups,
4. bookkeeping information about free and busy resources,
5. execution of the PolGrawAllSky code.

After the MPI environment is successfully initialized, the virtual grouping execution model is carried on and the master-slave model becomes the basic architecture of the simulation. The computing resources are split into non-equal parallel groups to ensure that the execution of all embedded PolGrawAllSky codes takes approximately the same time, in order to achieve proper load balance. The implementation for global information exchange in bookkeeping is based on MPI all-to-all non-blocking communication.

3.1.1. Domain decomposition

MPP with internal scheduling and work-flow may be prone to problems of load balancing, especially when many parallel computations are embedded into one big run. This typically results in limiting scalability. Therefore, to optimally use big MPP systems with more than 10 000 parallel tasks, we have implemented a domain decomposition algorithm based on estimation of optimal number of parallel tasks per frequency band, as illustrated in Fig. 5.

The execution of skyfarmer starts with reading various parameters like frame number, input data directory, output data directory and two input files *cpuperband.dat* containing two columns - frequency f_k and the corresponding estimation for optimal number of parallel tasks $S(f_k)$, as well the file *frequency.dat* containing the list of frequency bands to be analyzed (see the Algorithm 1). These files are used to compute group size $S(f)$ for a given frequency band f and generate a set of encapsulated parallel runs. The size of group of tasks to be used by various bands is estimated using first order spline-polynomial interpolation

$$S(f) = a \cdot f + b \quad = \underbrace{\frac{S_2 - S_1}{f_2 - f_1}}_{a} \cdot f + \underbrace{S_1 - \frac{S_2 - S_1}{f_2 - f_1} \cdot f_1}_{b = S_1 - a \cdot f_1}, \quad (2)$$

where f_1 and f_2 are two frequencies in the *cpuperband.dat* file neighboring f and $S(f_1)$ and $S(f_2)$ are the corresponding numbers of parallel tasks. In our tests we used an estimation presented in Fig. 5 and generated on the basis of different scalability tests on a computing cluster with up to 50 000 computing cores with 10 GigaFLOPS performance. This estimation may of course be changed or recomputed if the load balance for each frequency band needs to be different than 1 hour or amount of available parallel tasks as well as when the performance is different. For instance, the values for the CPU per band could be reduced to perform the computation longer than in 1 hour.

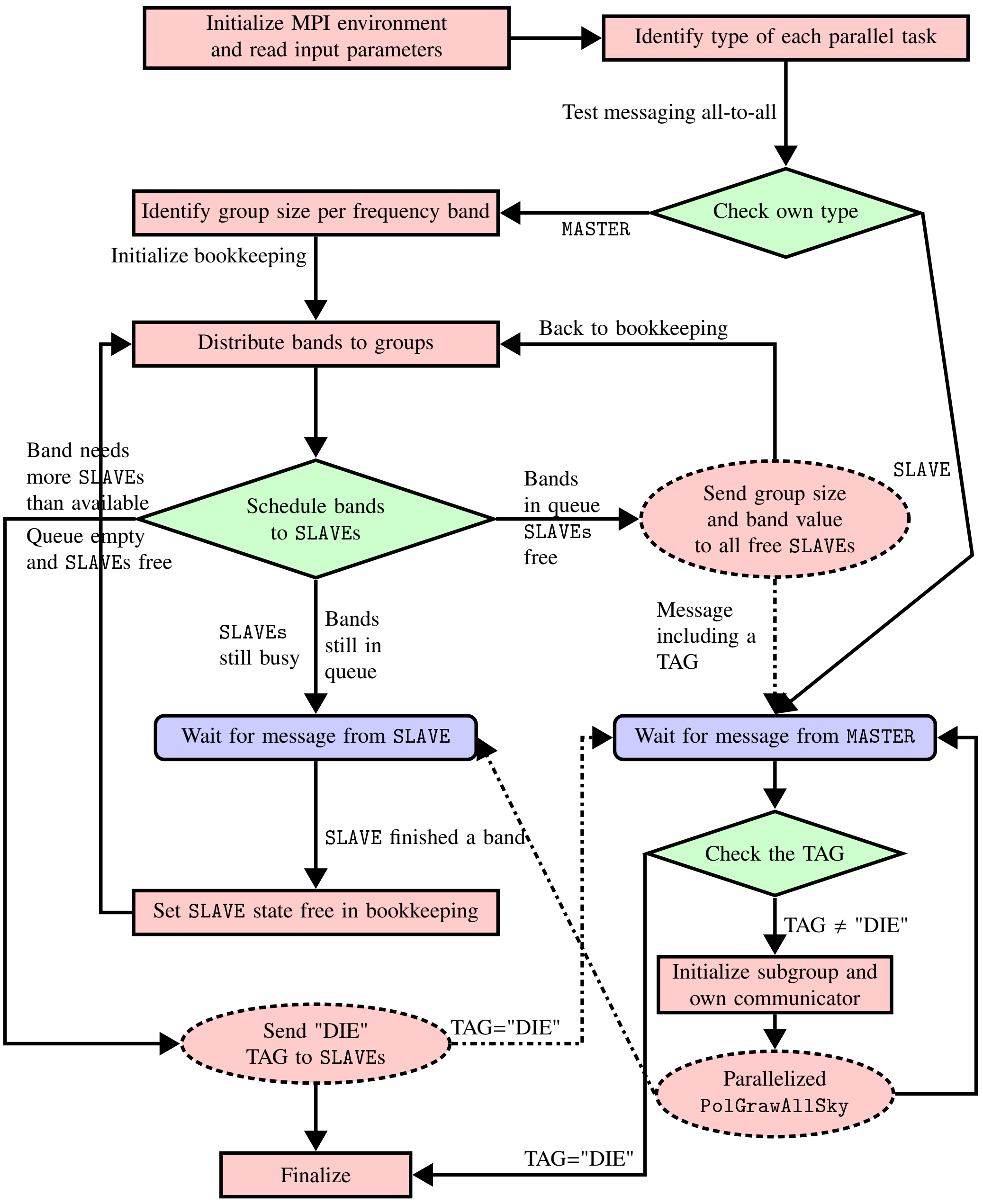
List of bands in the file *frequency.dat* could be given in any order and will be sorted internally by the skyfarmer in such a way that the computing intensive cases are performed first. This procedure aids to correctly estimate the resources - number of tasks to be divided into virtual groups and ran with the encapsulated PolGrawAllSky code. If any of the frequency bands would require all available resources, then only this band will run. The corresponding virtual group will occupy all the resources and the search of other frequency bands will not be performed until the first batch of jobs is finished. This could be counted as an almost sequential run even if each search at a given frequency would be running in parallel.

In the current version of the skyfarmer a case of global insufficiency appears and no simulation for frequencies given in *frequency.dat* will be carried out when the number of available resources is less than is necessary for the highest frequency case. So the estimate based on *cpuperband.dat* must allow to compose virtual groups for each frequency given in *frequency.dat*, even if this group will occupy the whole available resources.

3.1.2. Bookkeeping algorithm - scheduling for embedded parallelization

For advanced scalability one always needs to provide more resources than the summary need of all the tasks for each frequency search. In this way all embedded PolGrawAllSky codes may run in parallel and optimal speed-up i.e., the reduction of the *wall time* of execution will be reached. Alas, it is not always possible to provide as much resources as it is needed to complete all computation in parallel at once, therefore some sub-sequential runs of groups are organized by the scheduling and bookkeeping mechanism implemented in skyfarmer .

The MASTER process initializes a bookkeeping algorithm that keeps the information about the slave tasks' status and tracks the subgroups in queue, allowing them to run as soon as enough resources are available. During a run the MASTER stores the status of SLAVE processes as *busy* or *free* in a *book* array, the index of which corresponds to initial rank of task. At start all are initialized as free except the MASTER task, which is always marked busy. Subsequently, the MASTER distributes the information about the defined groups with corresponding band frequency to SLAVEs, switches their corresponding states in bookkeeping array into busy and puts into infinite waiting state expecting messages from any SLAVE. As soon as a message tag "FINISHED" is received from a given SLAVE, the status of all ranks in group, whom the slave belongs to are changed to free. Following, a



Initialize MPI environment and read input parameters
Identify type of each parallel task
Test messaging all-to-all
Check own type
MASTER
SLAVE
Identify group size per frequency band
Initialize bookkeeping
Distribute bands to groups
Back to bookkeeping
Band needs more SLAVEs than available
Schedule bands to SLAVEs
Queue empty and SLAVEs free
Bands in queue SLAVEs free
Send group size and band value to all free SLAVEs
SLAVEs still busy
Bands still in queue
Message including a TAG
Wait for message from SLAVE
Wait for message from MASTER
SLAVE finished a band
Check the TAG
Set SLAVE state free in bookkeeping
TAG ≠ "DIE"
Initialize subgroup and own communicator
Send "DIE" TAG to SLAVEs
TAG="DIE"
Parallelized PolGrawAllSky
Finalize
TAG="DIE"

Figure 6: Flow diagram of the `skyfarmer` for running `PolGrawAllSky` at different frequency bands in separate MPI worlds/groups. Bricks represent usual algorithmic steps; diamonds are steps with decisions about future direction of run; ellipses are steps when message passing take place represented as dash-dotted paths; blocks with rounded corners represent steps in which `MASTER` or `SLAVE` ranks are waiting until a new MPI message is received.

check of the status of all the SLAVEs maintained in the structure book is made, and as a result the additional `free_ranks` array is populated with the ranks of slaves found in bookkeeping array stated as free. The size of the `free_ranks` array is used to identify any other frequency computation waiting in the queue with a group size that may fit the available free resources.

However, if meanwhile some resources were freed, the creation of a new communicator for a new group to start a new parallel simulation in addition to the already running ones is not possible due to use of collective mechanism for communicator creation in current version of `skyfarmer` - one needs to send the information about a new group redistribution of tasks to all tasks and this is possible only when all tasks are in the free state. In order to be able to create a new virtual group we have used `MPI_Comm_create`, which should be called simultaneously by all MPI ranks belonging to parent communicator. The SLAVES whose rank are not present in the new group are receiving the MPI_COMM_NULL signal.

As a result, one has to check at every new iteration if all the ranks and groups have finished their partial simulation and are in status free, before starting any new set of parallel groups. This limitation results in a loss of scalability - the waiting time for other unfinished parallel simulations (even if there are enough free resources available for a new simulation) may be decreased by using the *intercommunicators* (Dinan et al., 2011). Herewith the communicator creation of virtual groups is collective over those processes only that will be members in the resulting communicator. Disadvantage of the non-collective communicator creation is the *group creation cost* - time spent to create a new group increases exponentially depending on the number of parallel processes, because of the recursive merging nature of the algorithm. This can be amortized however by a potential benefit to particular application and hence the benefit for using it in `skyfarmer` should be investigated.

3.1.3. Algorithm for execution of embedded code in parallel

Like the MASTER, the SLAVEs are placed in an infinite loop which breaks only when the whole simulation is finished, i.e., when it reaches the box Finalize in Fig. 6. Initially, the SLAVEs are in the waiting state. When the first receive statement comes, it contains not only the particular information about groups to run, but also a tag. In the case of "DIE" tag, the infinite loop will be broken, otherwise it means there is a group, in which the SLAVE participates in a joint simulation. After receiving the membership of a respective group, all the SLAVEs place the communicator creation call with respect of their group; if the SLAVE does not find itself as a member of any group, then the group creation call returns MPI_COMM_NULL, and such SLAVE is being marked "FREE" to MASTER.

When the SLAVE identifies itself as a member of one of the groups, then it receives the respective frequency band, passed further to the embedded PolGrawAllSky code. After completing the execution the group frees their communicator and a "FREE" status tag is sent to the MASTER from all the SLAVEs members of the group. The cycle continues as long as all the bands are completed and the "DIE" tag is received by the MASTER. This is summarized by the Algorithm 2.

Algorithm 1: Algorithm for initialization, domain decomposition and bookkeeping

```
Data: MASTER rank
Result: Preparation of parallel simulations
Read and sort the band table;
Initializing the bookkeeping;
Define the domains as groups;
while infinitely do
    Check and count free slaves;
    if not enough free slaves then
        go into waiting state
    end
    Check bookkeeping for bands to run;
    if all groups fit then
        for each group do
            Send group size and band value to slaves;
            Create new communicators;
            Set the slaves as busy;
            Set the groups as submitted;
        end
    else
        if biggest group does not fit then
            Stop simulation;
        else
            if SLAVEs are busy then
                Go into waiting state;
                Finished message received;
                Set slaves free;
            else
                Finish simulation
            end
        end
    end
end
```

3.2. Implementing parallel Input/Output into the embedded code

At large parallelization scale, the limitations of codes based on trivial farm skeleton paradigms are inevitable, leading to the loss of performance. It is usually due to the Input/Output (I/O) activity rising up as "bottlenecks". The execution of embedded code on MPP systems with more than 1000 processing units faces such limitations of parallel file systems. Even if the encapsulated code is rarely using I/O, the performance of the farm is limited directly by the scalability of the file system. The fact that each parallel-running embedded code writes its results as separate files results in intensive usage of data storage. The final results are distributed in as many separate files as there are tasks. Handling of large number of files is then an additional problem in any distributed computing systems.

To eliminate this problem we have implemented a parallel writing mechanism to join the outputs (files with candidate signals). The candidate parameters are initially stored in the local memory of parallel SLAVEs running different sky loops and at the end of computation the data are written in a single file

Algorithm 2: Algorithm for running parallel embedded codes

```
Data: SLAVE rank
Result: Generation of parallel simulations
while infinitely do
    Receive information about all groups and status tag;
    if status tag is not DIE then
        for check all group do
            Receive band frequency;
            Receive particular group size;
            Receive ranks of group;
            Initialize and try to create group
            communicator;
            if New communicator is one whom slave must
            belong to then
                Save the group size, band and
                communicator information
            end
        end
        if New communicator is found to whom SLAVE
        must belong to then
            Call the embedded code using identified
            communicator, band frequency;
            Free particular group and communicator used
            from PolGrawAllSky code
        end
        Send FREE status to MASTER
    else
        Finish calculation
    end
end
```

per frequency band and frame, using the concept of collective operations supported by MPI `I/O`. This functionality is implemented at various levels of the embedded code by using joint MPI files that are created at the start of each embedded computation. Each parallel process obtains its part of the file for writing in the data of the candidate signals found in its analysis.

`I/O` algorithm starts with the acquisition of its own view of joint MPI file by each process participating in the operation. A view is defined in terms of three parameters: a displacement or location in the file given by the number of bytes from the beginning of the file, an elementary data type and a file type. Before the given process begins writing into joint MPI file in parallel, an `MPI_Allgather` collective communication is used to count signals that every `SLAVE` has computed and stored in its local memory. This allows each `SLAVE` to calculate the offset from which the `SLAVE` starts writing. The offset is obtained by counting the number of 5 doubles that every other process with lower rank has to write for each identified signal.

4. Performance and scalability analysis

Productive usage of any parallel code on a supercomputing system with thousands of tasks triggers a multitude of challenges significantly different from those which rise when running the same code in a sequential way, even if the amount of sequential executions is the same. When comparing a high throughput computation with sequential runs, the benefit of using parallelized code on massively parallel systems is the performance achieved by smaller time necessary to accomplish each parallel run. It may however be limited due to bottlenecks or inherently sequential parts of the parallelized application (Amdahl's Law, Rodgers 1985).

Unavoidably, the present version of parallelized code for GW signal searches has a maximal number of tasks up to which a speed-up can be gained. For instance, even essentially eliminating non-scaling elements in the `PolGrawAllSky` code was not enough to reach high efficiency at tests with more than 1000 parallel tasks. At higher scales the loss of performance due to the `I/O` activities was inevitable independently on using multi-site or single-site parallel computing systems. Non-optimal `I/O` activities strongly limits the scalability of any code. To overcome this one has to implement a system for storing the data in memory or writing them out at some point of computation using the `I/O` libraries optimized for parallel file systems. The necessity of collective `I/O` implementation for the `PolGrawAllSky` code was described in the previous section.

Even after the elimination of the `I/O` bottleneck, the scalability of the parallelized `PolGrawAllSky` is limited due to the complexity of the algorithm. Depending on parameter space and the search frequency (Fig. 5), there is a maximum scalability at which the `PolGrawAllSky` code can be used in parallel. Particularly, using low coupling of partial analysis performed in the `PolGrawAllSky` code, we were able to keep interprocess communication limited and to reach higher speed-up with more than 30 000 cores for a highly complex and computationally-intensive cases at frequencies as high as 2300 Hz.

Hitherto, it is crucial to combine many runs of the `PolGrawAllSky` code, when performing searches at different frequencies in one job, to extend the performance and scalability of the whole computation to levels above 30 thousand parallel tasks. By embedding the parallelized `PolGrawAllSky` into `skyfarmer` and implementing the domain decomposition and efficient bookkeeping algorithm described in section 3.1, we were able to reach scalability as high as 50176 parallel tasks for a test run when searching GW signal for 11 different frequencies [100,200,300,400,600,800,1000,1400,1700,2000,2300] in one massive parallel run.

To estimate efficiency of the parallelized `PolGrawAllSky` as well as of the `skyfarmer` code we have made a performance analysis and scalability tests using the CRAY XE6 supercomputer at the High Performance Computing Center Stuttgart (HLRS), called Hermit [4]. This cluster consists of 3552 nodes interconnected through an InfiniBand network with 113 664 compute cores altogether and 1 petaFLOPS peak performance.

4.1. Strong and weak scaling

By running a fixed-size problem on a varying number of processors one can see how the timing of the computation scales

[4]https://www.hlrs.de/systems/platforms/cray-xe6-hermit

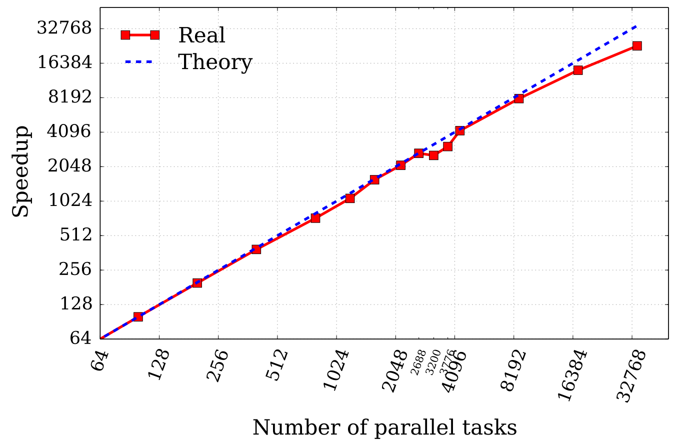


Figure 7: Strong scaling of the `skyfarmer` for joint computation of 7 frequency bands at once, using different amount of parallel tasks.

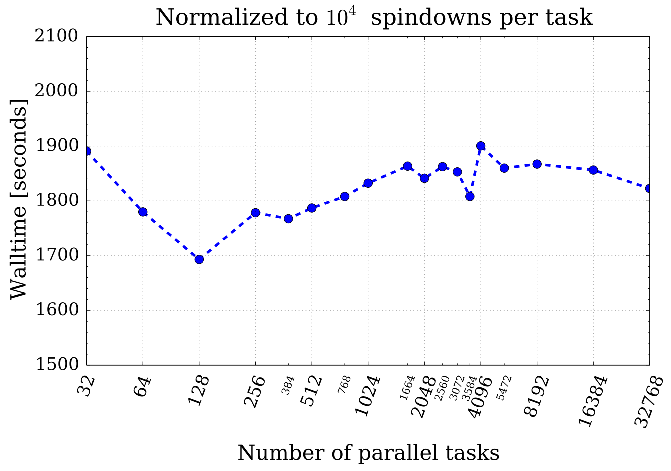


Figure 8: Weak scaling of the embedded `PolGrawAllSky` code. Each task involves a search of 10^4 spindowns. Average wall time to perform a task is shown for an increasing number of parallel tasks.

with the number of processors and estimate what is the part of the code that is efficiently parallelized. This feature is called the *strong scaling*. Fig. 5 shows the scalability of the embedded `PolGrawAllSky` code and speed-up tests: reduction of the number of CPU hours for the same-size simulations by increasing the number of parallel tasks, performed with the `skyfarmer` are presented in Fig. 7.

By increasing the complexity of the problem by adding frequency bands to the same computation we were able to reach high scalability. In each run we have analyzed 7 frequency bands ranging from 100 to 700 Hz, with a 100 Hz step. The results show that our current implementation that uses MPI is able to run with up to 32 000 tasks in parallel very efficiently.

However, as the number of computations of spindowns for each sky position per parallel task decreases, the scaling is detached from the ideal linear speed-up, because the communication per task starts to dominate the computation time. Also, not optimal algorithms of the scheduling and domain decomposition of current version of `skyfarmer` degrade the performance. In other words the problem is not "heavy" enough to keep bigger amount of parallel processes "busy" with computation, as well as the mechanism of the group size estimation per chosen frequency for running parallel MPI worlds, described in section 3.1, is not fully optimal yet.

Secondly, the available tasks were not always perfectly fitting into the number of parallel tasks necessary for running a given 7 frequency bands as the domain size per frequency is estimated by first order spline-polynomial interpolation (see Eq.2) and must be rounded down to the nearest integer. As a result, some tasks were simply not used. This is visible in Fig. 7 when the speed-up "flattens out" at around 4000 tasks, but picks up again when available tasks are using all the available resources.

The strong scaling test shows the advantages and disadvantages of using bookkeeping and the domain decomposition mechanism. On one hand they are enabling the large computation even if not enough resources are provided to analyze all frequencies in parallel at once, but on the other hand the optimal speed-up is reached only if the optimization mechanisms balance the needed and available resources.

To determine how large a problem size could be we performed a weak scaling test by fixing the amount of work per processor - the ratio of spindown loop computations to number of tasks - and compared the execution time. Fig. 8 shows the average time spent by a task consisting of analysis of 10^4 spindown loops as a function of the number of tasks on which the computation is performed. This illustrates the fact that the combination of tasks at all scales, from 32 till 16384 tasks, into one parallel computation is efficiently realizable with time scales of around 1900 seconds per frequency. We were able to test it by computing 17 frequencies (from 100 till 2000) at once in only 56 minutes on 50272 tasks e.g., a half of the Cray Hermit supercomputer.

4.2. Load balancing

For an efficient use of the computational time it is essential that all the cores finish their work as synchronously as possible. Locally on each processor, this is done automatically since the coding and optimizing is made to reach maximal speed-up by using the `round-robin` algorithm. FFT computations consisting of two loops over the sky positions, an inner loop over frequency derivative and finally the FFT that evaluates $\mathcal{F}$-statistic on a grid of frequencies were distributed in parallel in a fully decoupled manner. Time scales for each parallel computation were approximately the same as it can be seen can in Fig. 9. The unbalanced part at some sky positions consist of computations that were about 30% faster. The number of these exceptionally "light" computations is less than 1%. A possible non-optimal usage of the computation power when some of the tasks spend more of their time in the waiting state is decreasing with the number of computations per band and the latency is "hidden" by the scheduling system. By keeping the problem size big enough the sub-optimal usage of resources due to unbalanced computation is practically eliminated.

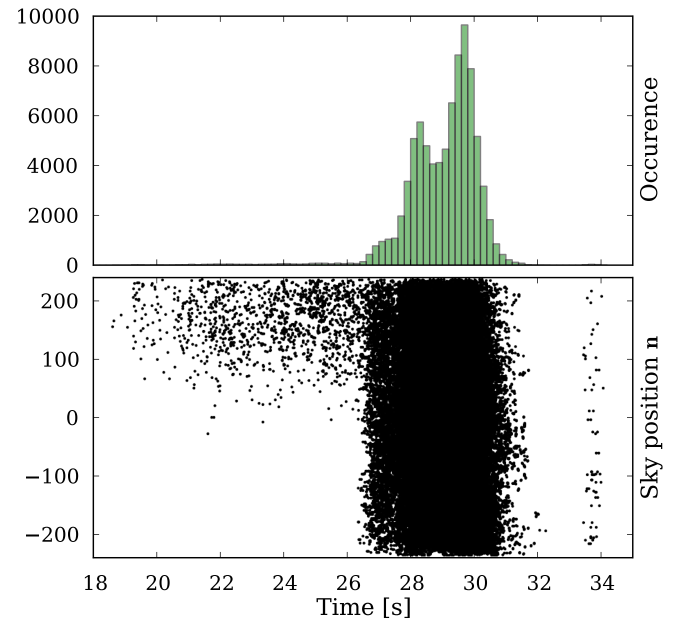


Figure 9: Relative distribution of computation for a search in each sky position, showing an almost balanced computation, spending on average the same amount of time per search. The unbalanced part is negligibly small.

5. Summary and Outlook

We have parallelized the `PolGrawAllSky` code, developed by the Polgraw-Virgo group for the search of the GWs from spinning neutron stars. The parallel version of the `PolGrawAllSky` embedded in `skyfarmer` proved to scale up sufficiently well up to 50 thousand CPU cores in one parallel job when performing the GW search for many different frequency bands at once.

The `I/O` activity showing up as a potential "bottleneck" in parallelized `PolGrawAllSky` code for runs on over 1000 cores was neutralized by implementing a parallel `I/O` of MPI to generate join candidate signal files per each frequency band.

The implementation of `MPI_Groups`, MPI `I/O` and non-blocking global communications allowed to escape a massive communication overhead, limiting the code however for running on HPC clusters with highly efficient interconnects like InfiniBand.

We have re-engineered a non-optimal memory management algorithm of the `PolGrawAllSky` code since it initially was developed for sequential runs and was causing "memory leaks" when running as an embedded part of the `skyfarmer` on massively parallel systems.

Using `MPI_Groups` for organizing a farm skeleton allowed us to incorporate the parallel version of the `PolGrawAllSky` code into a `skyfarmer` without a need of its global re-engineering.

The benefit for using non-collective communicator creation mechanism in `skyfarmer` need to be still investigated.

In order to keep up with the huge size of the parameter space for analyzing the future data of GW detector experiments, a hybrid parallelization of `skyfarmer` is necessary. Here, the computation, e.g. FFT in `PolGrawAllSky` code, could be partially driven on Graphical Processor Units (GPU) or other hardware accelerators.

Hybrid parallelization of the `PolGrawAllSky` and more intelligent scheduling and bookkeeping in task farming mechanism of the `skyfarmer` would also be crucial for optimal use of the future massively parallel exascale computing systems that are planned to be available around 2020, with more than 100 million CPUs and many multi-core architectures by combining processors and co-processors.

6. Acknowledgments

The identifying the bottlenecks and non-optimal usage of resources during simulations, were possible by using systems with more than 10 000 cores for testing the code on massively parallel system Cray XE6 Hermit at High Performance Computing Center HLRS in Stuttgart[5]. The developments on parallelizing `PolGrawAllSky` and designing the `skyfarmer` were supported by the Helmholtz program "Supercomputing" . M.B. and A.K. would to thank Steinbuch Centre for Computing of Karlsruhe Institute of Technology for support during the visits when this work was done. The work of M.B. and A.K was supported in part by the Polish Ministry of Science and Higher Education grant DPN/N176/VIRGO/2009 and National Science Center grants UMO-2013/01/ASPERA/ST9/00001 and 2012/07/B/ST9/04420.

References

Aasi, J., Abadie, J., Abbott, B. P., Abbott, R., Abbott, T. D., Abernathy, M., Accadia, T., Acernese, F., Adams, C., Adams, T., & et al. (2013a). Einstein@Home all-sky search for periodic gravitational waves in LIGO S5 data. Phys. Rev. D, *87*, 042001.

Aasi, J., Abadie, J., Abbott, B. P., Abbott, R., Abbott, T. D., Abernathy, M., Accadia, T., Acernese, F., & et al. (2013b). Prospects for Localization of Gravitational Wave Transients by the Advanced LIGO and Advanced Virgo Observatories. *ArXiv e-prints*, .

Aasi, J., Abadie, J., Abbott, B. P., Abbott, R., Abbott, T. D., Abernathy, M., Accadia, T., Acernese, F., & et al. (2014). ... *ArXiv e-prints*, .

Astone, P., Borkowski, K. M., Jaranowski, P., Piętka, M., & Królak, A. (2010). Data analysis of gravitational-wave signals from spinning neutron stars. V. A narrow-band all-sky search. Phys. Rev. D, *82*, 022005.

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge.

Cole, M. (2004). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, *30*, 389 – 406.

Dinan, J., Krishnamoorthy, S., Balaji, P., Hammond, J. R., Krishnan, M., Tipparaju, V., & Vishnu, A. (2011). *Noncollective communicator creation in MPI* volume 6960 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

Einstein, A. (1916). Näherungsweise Integration der Feldgleichungen der Gravitation. *In: Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften Berlin*, (p. 688).

Ernsting, S., & Kuchen, H. (2012). Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, *7*, 129–138.

Gonzalez-Velez, H., & Leyton, M. (2010). A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software - Practice and Experience*, *40*, 1135–1160.

[5]Project ACID 12863

Gropp, W., & Lusk, E. (1995). Dynamic process management in an mpi setting. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on* (pp. 530–533).

Hulse, R. A., & Taylor, J. H. (1975). . *Astroph. J*, *195*, L51.

Jaranowski, P., Królak, A., & Schutz, B. F. (1998). Data analysis of gravitational-wave signals from spinning neutron stars: The signal and its detection. Phys. Rev. D, *58*, 063001.

Kleinrock, L. (1964). Analysis of a time-shared processor. *Naval Research Logistics Quarterly*, *11*, 59–73.

Message Passing Interface Forum (2012). *MPI: A Message-Passing Interface Standard, Version 3.0.* Specification.

Rodgers, D. P. (1985). Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, *13*, 225–231.

Saulson, P. R. (1994). *Fundamentals of Interferometric Gravitational Wave Detectors*. Singapore: World Scientific.