Balancing load on a multiprocessor system with event-driven approach

Alexander Degtyarev and Ivan Gankevich

Saint Petersburg State University, Saint Petersburg 199034, Russia
deg@csa.ru, i.gankevich@spbu.ru

Abstract. There are many causes of imbalanced load on a multiprocessor system such as heterogeneity of processors, parallel execution of tasks of varying complexity and also difficulties in estimating complexity of a particular task, however, if one can treat computer as an event-driven processing system and treat tasks as events running through this system the problem of load balance can be reduced to a well-posed mathematical problem which further simplifies to solving a single equation. The load balancer measures both complexity of the task being solved and performance of a computer running this particular task so that a load distribution can be adjusted accordingly. Such load balancer is implemented as a computer program and is known to be able to balance the load on heterogeneous processors in a number of scenarios.

Keywords: load balance, event-driven architecture, heterogeneous system, multiprocessor computer

Introduction

Load balance is maintained by adjusting distribution of computational tasks among available processors with respect to their performance, and inability to distribute them evenly stems not only from technical reasons but also from peculiarity of a problem being solved. On one hand, load imbalance can be caused by heterogeneity of the tasks and inability to estimate how much time it takes to execute one particular task compared to some other task. Such difficulties arise in fluid mechanics applications involving solution of a problem with boundary conditions when the formula used to calculate boundary layer differs from the formula used to calculate inner points and takes longer time to calculate; the same problem arises in concurrent algorithms of intelligent systems that have different asymptotic complexities but solve the same problem concurrently hoping to obtain result by the fastest algorithm. On the other hand, load imbalance can be caused by heterogeneity of the processors and their different performance when solving the same problem and it is relevant when tasks are executed on multiple computers in a network or on a single computer equipped with an accelerator. Therefore, load imbalance can be caused by heterogeneity of tasks and heterogeneity of processors and these peculiarities should be both taken into account to maintain load balance of a computer system.

From mathematical point of view, load balance condition means equality of distribution function F of some task metric (e.g. execution time) to distribution function G of some processor metric (e.g. performance) and the problem of balancing the load is reduced to solving equation

$$F(x) = G(n), \tag{1}$$

where x is task metric (or time taken to execute the task) and n is processor metric (or relative performance of a processor needed to execute this task). Since in general case it is impossible to know in advance neither the time needed to execute the task on a particular processor, nor the performance of a processor executing a particular task, stochastic approach should be employed to estimate those values. Empirical distribution functions can be obtained from execution time samples recorded for each task: task metric is obtained dividing execution time by a number of tasks and processor metric is obtained dividing a number of tasks by their execution time. Also, any other suitable metrics can be used instead of the proposed ones, e.g. the size of data to be processed can be used as a task metric and processor metric can be represented by some fixed number.

It is easy to measure execution time of each task when the whole system acts as an event-driven system and an event is a single task consisting of program code to be executed and data to be processed. In this interpretation, load balancer component is connected to a processor recursively via profiler forming feedback control system. Profiler collects execution time samples and load balancer estimates empirical distribution functions and distributes new tasks among processors solving equation 1.

Static load balancing is also possible in this event-driven system and for that purpose a set of different load balancers can be composed into a hierarchy. In such hierarchy, distribution function is estimated incrementally from bottom to top and hierarchy is used to maintain static load balance. Physical processors are composed or decomposed into virtual ones grouping a set of processors and assigning them to a single load balancer or assigning one physical processor to more than one load balancer at once. So static load balancing is orthogonal to dynamic load balancing and they can be used in conjunction.

To summarize, recursive load balancing approach targets problems exhibiting not only dynamic but also static imbalance and the balance can be achieved solving a single equation.

Related work

The main drawback of existing parallel programming technologies is their inability to perform load balancing across different computing devices. Each device is associated with a different type of a workload, e.g. disk is associated with I/O and processor with pure computations. Although, almost any program involves computations and reading/writing data to disk, today's standards for multi-core programming (like OpenMP [2]) do not allow to do it in parallel — there are no pipelines neither in OpenMP standard nor in any emerging technology known

to date. Moreover, there are many other computing devices that can benefit from pipelines — mainly network interface cards and GPUs — to perform computations and data transfer simultaneously. So, modern parallel programming technologies do not allow to co-exist different type of workloads in a single program, but many programs may benefit from it, exploiting additional degrees of parallelism.

In contrast to parallel programming technologies, event-driven approach allows to use every device in the computer in a unified way, and easily form a pipeline between different devices. Event-driven architecture have been used extensively to create desktop applications with graphical user interface since MVC paradigm [11] was developed and nowadays it is also used to compose enterprise application components into a unified system with message queues [9, 13], however, it is rarely implemented in scientific applications. One example of such usage is GotoBLAS2 library [7, 8]. Although, it is not clear from the referenced papers, analysis of its source code[1] shows that this library uses specialized server object to schedule matrix and vector operations' kernels and to compute them in parallel. The total number of CPUs is defined at compile time and they are assumed to be homogeneous. There is a notion of a queue implemented as a linked list of objects where each object specifies a routine to be executed and data to be processed and also a number of CPUs to execute it on. Server processes these objects in parallel and each kernel can be executed in synchronous (blocking) and asynchronous (non-blocking) mode. So, compared to event-driven system GotoBLAS2 server uses static task scheduling, its tasks are not differentiated into production and reduction tasks, both the tasks and the underlying system are assumed to be homogeneous. GotoBLAS2 library exhibits competitive performance compared to other BLAS implementations [7, 8] and it is a good example of viability of event-driven approach in scientific applications. Considering this, event-driven system can be seen as a generalization of this approach to a broader set of applications.

There are a number of research works in which the authors develop systems borrowing some features from event-driven approach. For example, in [10,12,14] a concurrent object-oriented system similar to our system is described. In contrast to our system, it uses messages rather than events to transfer data, and it does not allow load-balancing across different computing devices. Another example is [1], in which the author describes a system which uses *supervisor trees* to organize concurrent processes and manage resources. These structures are similar to hierarchies, but we have different hierarchies for servers and tasks, so that system resources and the flow of computations can be managed separately. So, there are some works which borrow different parts of a typical event-driven system, however, they either do not put emphasis on load-balancing, or do not describe their system as an event-driven, thus not exploiting full benefits of it.

Load-balancing is one of the main tasks of an operating system, and in our view in high-performance computing it should be delegated to some intermediate software layer which lies between operating system and application. So, in con-

[1] Source code is available in https://www.tacc.utexas.edu/tacc-projects/gotoblas2/.

trast to described approaches it would be useful to implement full event-driven system and hide message passing and synchronization logic in it.

1 Implementation of event-driven system

The whole system was implemented as a collection of C++ classes, and problem-solving classes were separated from utility classes with an event-driven approach. In this approach, problem solution is represented by a set of executable objects or "employees", each implementing a solution of one particular part of a problem. Each executable object can implement two methods. With the first method employee either solves part of the problem or produces child executable objects (or "hires" additional employees) to delegate problem solution to them. Since upon completion of this method no object is destroyed, it is called "production" task or "upstream" task as it often delegates problem solution to a hierarchy layer located farther from the root than the current layer is. The second method collects execution results from subordinate executable objects and takes such object as an argument. Upon completion of this method the child object is destroyed or "fired" so that the total number of executable objects is reduced. Hence this task is called "reduction" task or "downstream" task since the results are sent to a hierarchy layer located closer to root. Executable objects can send results not only to their parents but also to any number of other executable objects, however, when communication with a parent occurs the child object is destroyed and when the root object tries to send results to its nonexistent parent, the program ends. Execution of a particular object is performed via submitting it to a queue corresponding to a particular processor. Child and parent objects are determined implicitly during submission so that no manual specification is needed. Finally, these objects are never copied and are accessed only via their addresses. In other words, the only thing that is required when constructing an executable object is to implement a specific method to solve a task and object's life time is implicitly controlled by the system and a programmer does not have to manage it manually.

Execution of objects is carried out concurrently and construction of an executable object is separated from its execution with a thread-safe queue. Every message in a queue is an executable object and carries the data and the code needed to process it and since executable objects are completely independent of each other they can be executed in any order. There are real server objects corresponding to each queue in a system which continuously retrieve objects from a queue and execute their production or reduction tasks in a thread associated with the server object. Production tasks can be submitted to any queue, but a queue into which reduction tasks can be submitted is determined by a corresponding parent object so that no race condition can occur. Since each processor works with its own queue only and in its own thread, processing of queues is carried out concurrently. Also, each queue in a system represents a pipeline through which the data flows, however, execution order is completely determined by the

objects themselves. So, executable objects and their methods model control flow while queues model data flow and the flows are separated from each other.

Heterogeneity of executable objects can cause load imbalance among different queues and this problem can be solved introducing imaginary (i.e. proxy) servers and profilers to aid in distribution of executable objects. Imaginary server is a server tied to a set of other servers and its only purpose is to choose the right child server to execute an object at.

In the simplest case, a proper distribution can be achieved with round-robin algorithm, i.e. when each arriving object is executed on the next server, however, in general case, some additional information about completed runs is needed to choose the right server and this information can be collected with pluggable profiler objects. When a new object arrives to an imaginary server, actual profiling information is collected from child servers and specified distribution strategy is used to delegate execution of an object to an appropriate server, and some static distribution strategy is also possible. So, imaginary servers together with distribution strategies and profilers can be used to distribute executable objects among real servers taking into account some profiling information of completed object executions.

The class diagram of the whole event-driven system is depicted in the Figure 1 and the system works as follows.

1. When a program execution starts, the hierarchy of imaginary and real servers is composed. All real servers are launched in a separate threads and processing of executable objects starts.
2. The first object is created and submitted to the imaginary server at the top of the hierarchy. The server employs specified distribution strategy to choose an appropriate server from the next layer of the hierarchy to send the object to. The profiler gathers measurements of completed runs from subordinate servers and decides where to send an object.
3. The previous step repeats until the bottom level of the hierarchy is reached and real server which was found with the distribution strategy starts execution of an object.
4. Object is executed and measurements are made by a profiler. If during execution more executable objects are created and submitted to the top imaginary server, the whole algorithm is repeated for each new object; if the root object submits reduction task then all servers in the hierarchy are shut down, and program execution ends.

To sum up, the whole system is composed of the two hierarchies: one hierarchy represents tasks and data and their dependencies employing executable objects, the other hierarchy represents processing system employing imaginary and real server objects. Mapping of the first hierarchy to the second is implicit and is implemented using message queues. Such composition allows easy configuration of dynamic and static load distribution strategies and allows programming with simple executable objects.

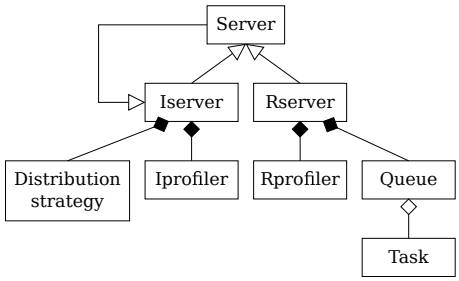


Fig. 1: Class diagram for an event-driven system. "Iserver" denotes imaginary server and "Rserver" denotes real server.

2 Implementation of distribution strategy

Recursive load balancing was implemented as a load distribution strategy, however, equation 1 was not solved directly. The first problem occurring when solving this equation directly was that task metric x cannot be computed before actually running the task so it was estimated to be an average metric of a number of previous runs. The second problem was that when the task metric is known, the result of direct solution of equation 1 is not an identifier of a processor to execute the task on but it is number n – relative performance of a processor needed to execute the task and the number n is not particularly useful when determining where to execute the task. Therefore, equation 1 was not solved directly but its main idea was realized in an algorithm similar to round-robin.

The resulting algorithm works as follows.

1. First, algorithm collects samples recorded by profilers of child servers as well as estimates task metric and processor metric using values from previous runs. At this stage, not only averaging but also any other suitable predicting technique can be used.
2. Then, probability of having a task with metric equal to computed task metric is determined by counting samples equal to computed task metric and dividing it by the total number of samples.
3. The cursor pointing at the processor to execute the next task on is incremented by a step equal to a product of computed probability and computed processor metric.
4. Then, by recursively subtracting metric of each processor from the cursor, the needed processor is found and the task is executed on it.

The resulting mathematical formula for each step can be written simply as

$$cursor = cursor + F(\bar{x})\bar{n},$$

where $\bar{x}$ is a task metric and $\bar{n}$ is a processor metric. In case of fully homogeneous system and all tasks having equal metric this algorithm is equivalent to round-robin: all processors have metric equal to 1 and probability is always 1 so that the cursor is always incremented by 1.

Although, the algorithm is simple, in practice it requires certain modifications and a robust profiler to work properly. Since algorithm balances reciprocal values of task metric t (execution time) and processor metric $1/t$ (processor throughput), even a slight oscillation of a task metric can affect the resulting distributions greatly. The solution to this problem is to smooth samples with a logarithm function and it can be done in a straightforward way, because the algorithm does not make assumptions about metrics' dimensions and treat them as numbers. The second problem is that the algorithm should be implemented with integer arithmetic only to minimize overhead of load balancing. This problem can be solved by omitting mantissa after logarithm is applied to a sample and in that case processor metric is equal to task metric but has an opposite sign. The last major problem is that the distribution of task metric may change abruptly during program execution, which renders samples collected by a profiler for previous runs useless. This problem is solved by detecting a sharp change in task execution time (more than three standard deviations) and when outliers are detected the profiler is reset to its initial state in which distribution is assumed to be uniform. As a result of applying logarithm to each sample the algorithm becomes unsuitable for relatively small tasks and for tasks taking too much time, and although such tasks are executed, the samples are not collected for them as they often represent just control flow tasks. To summarize, the modified algorithm is implemented using integer arithmetic only, is suitable for relatively complex tasks and adapts to a rapid change of a task metric distribution.

One problem of the algorithm that stands aside is that it becomes inefficient in the event of high number of tasks with high metric values. It happens because when task is assigned to a particular processor it is not executed directly but rather gets placed in a queue. If this queue is not empty the task can reside in it for such a long time that its assignment to a particular processor will not match actual distribution function. The solution is simple: these stale tasks can be easily detected by recording their arrival time and comparing it with the current time and when such tasks are encountered by a queue processor, they can be redistributed to match the current distribution function. However, an existence of stale tasks is also an evidence that the computer is not capable of solving a problem fast enough to cope with continuously generated tasks and it is an opportunity to communicate with some other computers to solve the problem together. From a technical point of view, delegation of tasks to other computers is possible because tasks are independent of each other and read/write (serialization) methods are easily implemented for each of them, however, the problem was not addressed herein, and only load redistribution within a single computer was implemented.

Described algorithm is suitable for distributing production tasks, but a different algorithm is needed to distribute reduction tasks. Indeed, when executable objects come in pairs consisting of the child and its parent, all children of the parent must be executed on the same server so that no race condition takes place, so it is not possible to distribute the task on an arbitrary server but a particular server must be chosen for all of the child tasks. One possible way of choosing

a server is by applying a simple hashing function to parent's memory address. Some sophistication of this algorithm is possible, e.g. predicting memory allocation and deallocation pattern to distribute reduction tasks uniformly among servers, however, considering that most of the reduction tasks in tested program were simple (the reason for this is discussed in Section 3.1) the approach seemed to be non viable and was not implemented. So, simple hashing algorithm was used to distribute reduction tasks among servers.

To summarize, recursive load distribution algorithm by default works as round-robin algorithm and when a reasonable change of task execution time is detected it automatically distributes the load in accordance with task metric distribution. Also, if there is a change in processor performance it is taken into account by relating its performance to other processors of computing system. Finally, if a task stays too much time in a queue it is distributed once again to match current distribution function.

3 Evaluation

Event-driven approach was tested on the example of hydrodynamics simulation program which solves real-world problem [3–6]. The problem consists of generating real ocean wavy surface and computing pressure under this surface to measure impact of the external excitations on marine object. The program is well-balanced in terms of processor load and for the purpose of evaluation it was implemented with introduced event-driven approach and the resulting implementation was compared to existing non event-driven approach in terms of performance and programming effort.

Event-driven architecture makes it easy to write logs which in turn can be used to make visualization of control flow in a program. Each server maintains its own log file and when some event occurs it is logged in this file accompanied by a time stamp and a server identifier. Having such files available, it is straightforward to reconstruct a sequence of events occurring during program execution and to establish connections between these events (to dynamically draw graph of tasks as they are executed). Many such graphs are used in this section to demonstrate results of experiments.

Generation of a wavy surface is implemented as a transformation of white noise, autoregressive model is used to generate ocean waves and pressures are computed using analytical formula. The program consists of preprocessing phase, main computer-intensive phase and post-processing phase. The program begins with solving Yule-Walker equations to determine autoregressive coefficients and variance of white noise. Then white noise is generated and is transformed to a wavy surface. Finally, the surface is trimmed and written to output stream. Generation of a wavy surface is the most computer-intensive phase and consumes over 80% of program execution time (Figure 7) for moderate wavy surface sizes and this time does not scale with a surface size. So, the program spends most the time in the main phase generating wavy surface (this phase is marked with $[G_0, G_1]$ interval in the graphs). The hardware used in the experiments is listed

in Table 2. The program was tested in a number of experiments and finally compared to other parallel programming techniques.

3.1 Evaluation of event-driven system

The first experiment consisted of measuring stale cycles and discovering causes of their occurrence. Program source code was instrumented with profiling directives and every occurrence of stale cycles was written to the log file. Also the total stale time was measured. Obtained results showed that stale cycles prevail in preprocessing and at the end of main phase but are not present in other parts of the program (Figure 2). The reason for this deals with insufficient amount of tasks available to solve during these phases which in turn is caused by global synchronizations occurring multiple times in preprocessing phase and naturally at the end of a program. Stale cycles in the main phase are caused by computation performing faster than writing results to disk: in the program only one thread writes data and no parallel file system is used. Further experiments showed that stale cycles consume at most 20% of the total execution time for 4 core system (Table 1) and although during this time threads are waiting on a mutex so that this time can be consumed by other operating system processes, there is also an opportunity to speed up the program. Considering file output performance stale cycles can only be reduced with faster storage devices combined with slower processors or with parallel file systems combined with fast network devices and interconnects. In contrast, the main cause of stale cycles in preprocessing phase deals with global synchronization and to minimize its effect it should be replaced by incremental synchronization if possible.

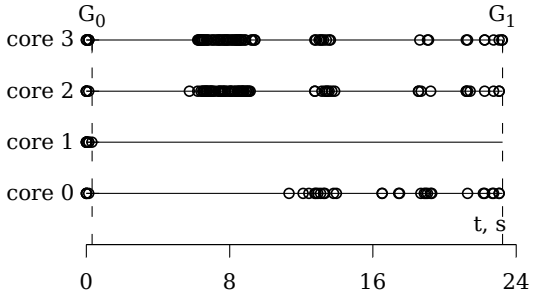


Fig. 2: Occurrences of stale cycles in preprocessing and at the end of the main computational phase of a program. Range $[G_0, G_1]$ denotes computationally intensive phase.

The next experiment consisted of measuring different types of overheads including profiling, load balancing, queuing and other overheads so that real performance of event-driven system can be estimated. In this experiment, the same technique was used to obtain measurements: every function causing overhead was

instrumented and also the total time spent executing tasks and total program execution time was measured. As a result, the total overhead was estimated to be less than 0.1% for different number of cores (Table 1). Also the results showed that reduction time is smaller than the total time spent solving production tasks in all cases (Table 1). It is typical of generator programs to spend more time solving data generating production tasks than solving data processing reduction tasks; in a data-centric program specializing in data processing this relation can be different. Finally, it is evident from the results that the more cores are present in the system the more stale time is introduced into the program. This behavior was explained in the previous experiment and is caused by imbalance between processor performance and performance of a storage device for this particular computational problem. To summarize, the experiment showed that event-driven system and recursive load distribution strategy do not incur much overhead even on systems with large number of cores and the program is rather code-centric than data-centric spending most of its execution time solving production tasks.

Classifier	Time consumer	Time spent, %		
		4 cores	24 cores	48 cores
Problem solution	Production tasks	71	33	19
	Reduction tasks	13	4	2
Stale time	Stale cycles	16	63	79
Overhead	Load distribution overhead	0.01	0.0014	0.0017
	Queuing overhead	0.002	0.0007	0.0005
	Profiling overhead	0.0004	0.0004	0.0003
	Other overheads	0.06	0.03	0.02

Table 1: Distribution of wall clock time and its main consumers in event-driven system. Time is shown as a percentage of the total program execution time. Experiments for 4 cores were conducted on the system I and experiments for 24 and 48 cores were conducted on the system II from Table 2.

In the third experiment, the total number of production tasks solved by the system was measured along with the total number of task resubmissions and it was found that there is high percentage of resubmissions. Each resubmission was recorded as a separate event and then a number of resubmissions for each task was calculated. The experiment showed that on average a total of 35% of tasks are resubmitted and analysis of an event log suggested that resubmissions occur mostly during the main computational phase (Figure 3). In other words 35% of production tasks stayed in a queue for too long time (more than an average time needed to solve a task) so underlying computer was not capable of solving tasks as fast as they are generated by the program. This result leads to a conclusion that if more than one computer is available to solve a problem, then there is a

natural way to determine what part of this problem requires multiple computers to be solved. So, high percentage of resubmissions shows that machine solves production tasks slower than they are generated by the program so multiple machines can be used to speed up problem solution.

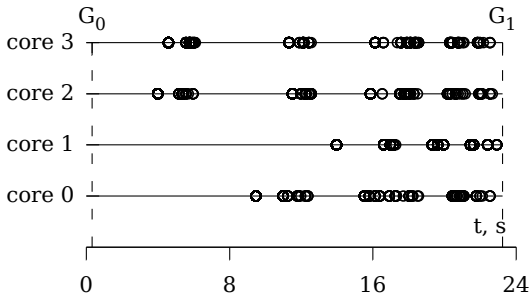


Fig. 3: Event plot of resubmission of production tasks staying in a queue for too long time. Range $[G_0, G_1]$ denotes computationally intensive phase.

In the final experiment overall performance of event-driven approach was tested and it was found to be superior when solving problems producing large volumes of data. In the previous research it was found that OpenMP is the best performing technology for the wavy ocean surface generation [3], so the experiment consisted of comparing its performance to the performance of event-driven approach on a set of input data. A range of sizes of a wavy surface was the only parameter that was varied among subsequent program runs. As a result of the experiment, event-driven approach was found to have higher performance than OpenMP technology and the more the size of the problem is the bigger performance gap becomes (Figure 4). Also event plot in Figure 5 of the run with the largest problem size shows that high performance is achieved with overlapping of parallel computation of a wavy surface (interval $[G_0, G_1]$) and output of resulting wavy surface parts to the storage device (interval $[W_0, W_1]$). It can be seen that there is no such overlap in OpenMP implementation and output begins at point W_0 right after the generation of wavy surface ends at point G_1. In contrast, there is a significant overlap in event-driven implementation and in that case wavy surface generation and data output end almost simultaneously at points G_1 and W_1 respectively. So, approach with pipelined execution of parallelized computational steps achieves better performance than sequential execution of the same steps.

Although OpenMP technology allows constructing pipelines, it is not easy to combine a pipeline with parallel execution of tasks. In fact such combination is possible if a thread-safe queue is implemented to communicate threads generating ocean surface to a thread writing data to disk. Then using *omp section* work of each thread can be implemented. However, implementation of parallel execution within *omp section* requires support for nesting *omp parallel* direc-

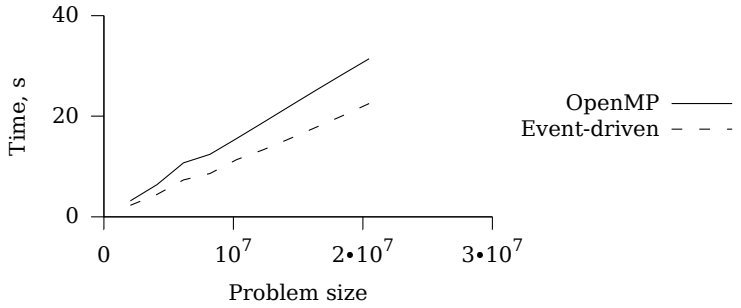


Fig. 4: Performance comparison of OpenMP and event-driven implementations.

tives. So, combining pipeline with parallel execution is complicated in OpenMP implementation requiring the use a thread-safe queue which is not present in OpenMP standard.

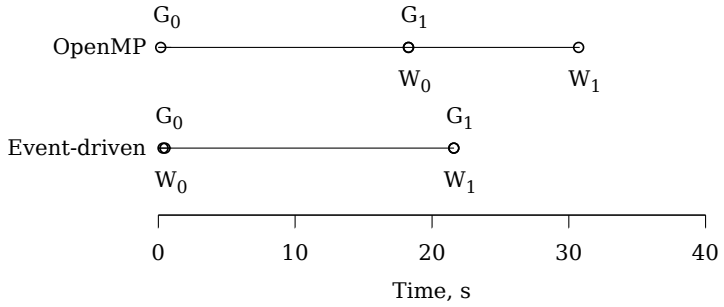


Fig. 5: Event plot showing overlap of parallel computation $[G_0, G_1]$ and data output $[W_0, W_1]$ in event-driven implementation. There is no overlap in OpenMP implementation.

To summarize, event-driven programming approach was applied to a real-world high-performance application and it was shown that it incurs low overhead, but results in appearance of stale periods when no problem solving is performed by some threads. The duration of these periods in the main phase can be reduced with faster storage equipment and the duration of stale periods in preprocessing phase can be reduced employing incremental synchronization techniques. Also, event-driven approach offers a natural way of determining whether program execution should scale to multiple machines or not, however, viability of such mode of execution was not tested in the present research. Finally, it was shown that event-driven approach is more efficient than standard OpenMP technology especially for large problem sizes and it was also shown that a pipeline combined with parallel execution works faster than sequential execution of parallelized steps.

3.2 Evaluation of load distribution strategy

Performance of recursive load distribution algorithm was compared to performance of round-robin algorithm and was tested in a number of scenarios with combinations of homogeneous and heterogeneous tasks and homogeneous and heterogeneous processors. In each experiment the total execution time and distributions of task metric and processor metric were measured and compared to uniform distribution case. All tests were performed on the same system (Table 2) and each scenario was run multiple times to ensure accurate results. Also, preliminary validation tests were performed to make sure that the algorithm works as intended. So, the purpose of evaluation was to demonstrate how algorithm works in practice and to measure its efficiency on a real problem.

Component	System	
Programming language	C++11	
Threading library	C++11 STL threads	
Atomics library	C++11 STL atomic	
Time measurement routines	`clock_gettime(CLOCK_MONOTONIC, ...)` `/usr/bin/time -f %e`	
Compiler	GCC 4.8.2	
Compiler flags	`-std=c++11 -O2 -march=native`	
	I	II
Operating system	Debian 3.2.51-1 x86_64	CentOS 6.5 x86_64
File system	ext4	ext4
Processor	Intel Core 2 Quad Q9650	2×Intel Xeon E5-2695 v2
Cores frequency (GHz)	3.00	2.40
Number of cores	4	24 (48 virtual cores)
RAM capacity (GB)	8	256
RAID device		Dell PERC H710 Mini
RAID configuration		RAID10
Storage device	Seagate ST3250318AS	4×Seagate ST300MM0006
Storage device speed (rpm)	7200	2×10000

Table 2: Testbed setup.

It has already been shown that the algorithm consumes only a small fraction of total execution time of a program (Table 1), so the purpose of the validation test was to show algorithm's ability to switch between different task metric distributions. The switching is performed when a significant change (more than three standard deviations) of a task execution time occurs. The test have shown that the switching events are present in preprocessing phase and do not occur in the main phase (Figure 7). The cause of the switching is a highly variable task execution time inherent to preprocessing phase. So, profilers' resets occur

only when a change of task execution time distribution is encountered and no switching is present when this distribution does not change.

The purpose of the first experiment was to show that the algorithm is capable of balancing homogeneous tasks on homogeneous computer and in that case it works like well-known round-robin algorithm. During the experiment, events of task submissions were recorded as well as additional profiler data and an event plot was created. In Figure 7a relative performance of each processor core is plotted and all the samples lie on a single line in the computational phase. Since this phase consists of executing tasks of equal metric, the straight line represents the uniform distribution of tasks among processor cores constituting round-robin algorithm. So, in the simplest case of homogeneous tasks and processors recursive load balancing algorithm works as round-robin algorithm.

The purpose of the second experiment was to show that recursive load-balancing algorithm is capable of balancing homogeneous tasks on heterogeneous processors and in that case it can distribute the load taking into account performance of a particular processor. Although natural application of such load balancing is hybrid computer systems equipped with graphical or other accelerators, the experiment was conducted by emulating such systems with a hierarchy of servers. It was found that load balancing algorithm can recognize performance of different components and adapt distribution of tasks accordingly (Figure 7b): I_1's first and second child servers have relative performance equal to 0.75 and 0.25 respectively whereas all children of I_2 server have relative performance equal to $\frac{1}{3}$. Also, this system setup shows performance similar to performance of the homogeneous computer configuration (Figure 6). So, recursive load balancing algorithm works on heterogeneous computer configurations and the performance is similar to homogeneous system case.

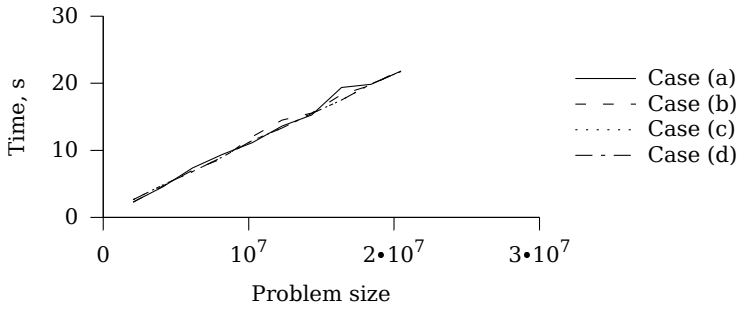


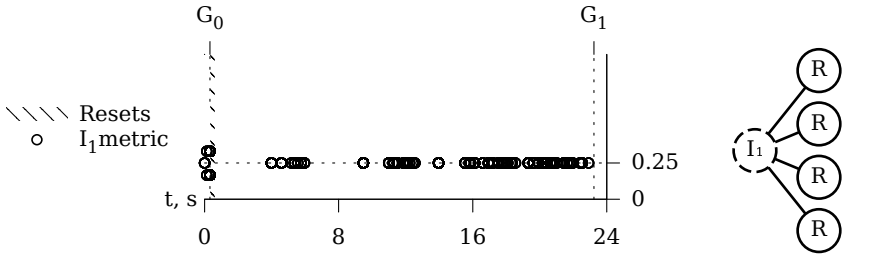
Fig. 6: Performance comparison of different server configurations. Configurations are listed in Figure 7.

The purpose of the third experiment was to show that the algorithm is capable of balancing heterogeneous tasks on a homogeneous system and the experiment showed that performance gain is small. For the experiment the source code generating a wavy surface was modified so that parts of two different sizes are

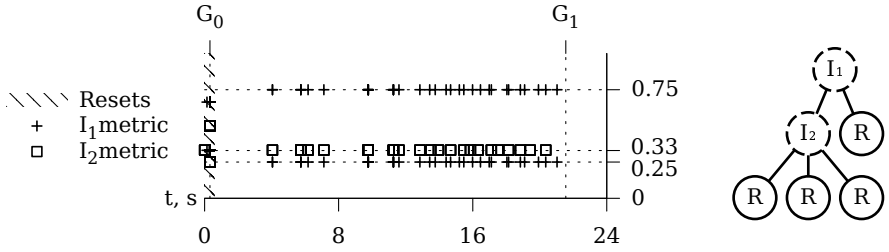
generated simultaneously. In order to balance such workload on a homogeneous system the step should be equal to $\frac{1}{2i}n, i = 1, 2, ...,$ where n is the processor metric (instead of being equal to 1 when parts have the same size) so that each processor takes two respective parts of the surface. In the Figure 7c showing results of the experiment the step reaches its optimal value of $\frac{1}{2}n$ (0.125 mark), however, it takes almost 8 seconds (or 40% of the total time) to reach this value. The first two cases do not exhibit such behavior and the step does not change during execution. Also, in the course of the experiment it was found that the step oscillates and to fix this it was smoothed with five point median filter and the number of samples was doubled. Finally, in subsequent experiments it was found that the more unique parts sizes are present in the main phase, the more samples should be collected to preserve the accuracy of the step evaluation, however, the increase in the number of samples led to slow convergence of the step to its optimal value. In other words, the more heterogeneous the tasks are, the more time is needed to find the optimal step value for them.

The purpose of the fourth and final experiment was to show that the algorithm is capable of balancing heterogeneous tasks on heterogeneous system and results were similar to the previous experiment. System configuration was the same as in the second experiment. Although, in the Figure 7d showing the results metrics and steps of both servers reach nearly optimal values, there are more disturbances in these processes. So, the algorithm works with heterogeneous tasks and system but heterogeneity of a system increases variability of the step. In other words, heterogeneity of a system also increases time needed to find the optimal step value.

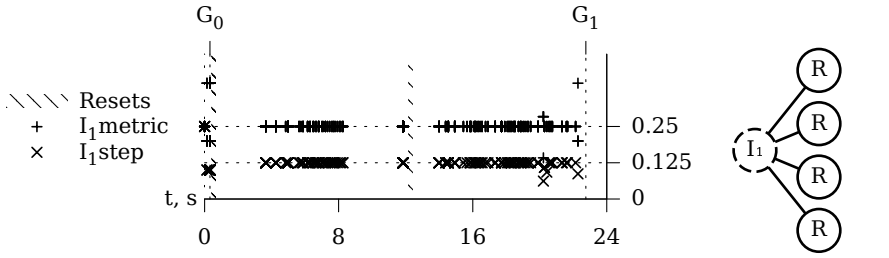
To summarize, from the experiments one can conclude, that the algorithm works on any system configuration and with any task combination, but requires tuning for a particular problem. However, experience obtained in the course of the experiments suggests that not only heterogeneity of tasks and computers increases the number of samples and convergence time but also there are certain task size distributions that cannot be handled efficiently by this algorithm and can extend this time indefinitely. One example of such distribution is linearly increasing task size. In this case step is always equal to $1/m$, where m is the number of samples, and there is no way to tune the algorithm to balance such workload. So, the downside of recursive load balancing algorithm is that it is suitable for closed metric distributions with low variability of the metric and more general and simple algorithms can be developed. Also, it is evident from the experiments from the Section 3.1 that in the tested program the dominating performance factor is balance between the speed of wavy surface generation and the speed of writing it to storage device. In that case, load balancing algorithm plays only a second role and any combination of computer and task heterogeneity demonstrates comparable performance as was depicted in Figure 6.



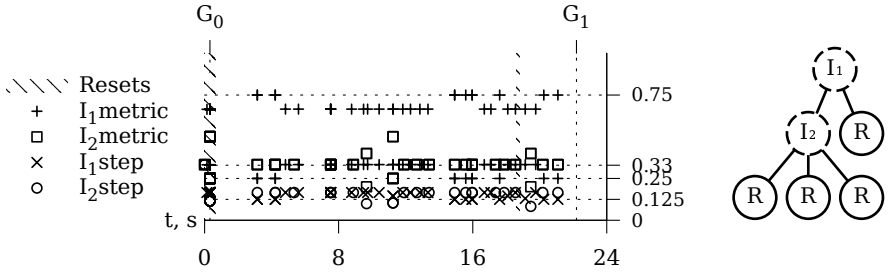
(a) Homogeneous tasks and homogeneous computer case.



(b) Homogeneous tasks and heterogeneous computer case.



(c) Heterogeneous tasks and homogeneous computer case.



(d) Heterogeneous tasks and heterogeneous computer case.

Fig. 7: Event plot of task submissions and relative performance of child servers recorded at the time of submissions. I denotes "Iserver" and R denotes "Rserver". Profiled servers are marked with dashed line.

Conclusions

The main advantage of event-driven approach is its applicability to both heterogeneous systems and heterogeneous tasks. This allows a programmer to rely on the technology to distribute the load on the processor cores evenly. Experiments showed that this approach works in a wide range of test cases and a real-world application. Moreover, in this application it performs better than popular OpenMP technology.

Apart from being more efficient than OpenMP the biggest advantage of event-driven approach is the ease of parallel programming. First of all, what is needed from a programmer is to develop a class to describe each independent task, create objects of that class and submit them to a queue. Programming in such a way does not involve thread and lock management and the system is flexible enough to have even the tiniest tasks executed in parallel. Second, relieving programmer from thread management makes it easy to debug this system. Each thread maintains its own log and any of both system and user events can be written to it and the sequence of events can be restored after the execution ends. Finally, with event-driven approach it is easy to write load distribution algorithm for your specific problem (or use an existing one). The only thing which is not done automatically is decomposition and composition of tasks, however, this problem requires higher layer of abstraction to solve.

The future work is to extend event-driven approach for distributed and hybrid (GPGPU) systems and to see if it is possible to cover those cases. The other possible direction of research is to create declarative language which acts as higher layer of abstraction and performs decomposition into tasks automatically.

Acknowledgments

Research was carried out using computational resources provided by Resource Center "Computer Center of SPbU"[2] and supported by Russian Foundation for Basic Research (project N 13-07-00747) and Saint Petersburg State University (projects N 9.38.674.2013, 0.37.155.2014).

References

1. Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors.* PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.
2. Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
3. A. Degtyarev and I. Gankevich. Wave surface generation using OpenCL, OpenMP and MPI. In *Proceedings of 8th International Conference "Computer Science & Information Technologies"*, pages 248–251, 2011.

[2] Official web site: http://cc.spbu.ru.

4. A. Degtyarev and I. Gankevich. Evaluation of hydrodynamic pressures for autoregression model of irregular waves. In *Proceedings of 11th International Conference "Stability of Ships and Ocean Vehicles", Athens*, pages 841–852, 2012.
5. A.B. Degtyarev and A.M. Reed. Modelling of incident waves near the ship's hull (application of autoregressive approach in problems of simulation of rough seas). In *Proceedings of the 12th International Ship Stability Workshop*, 2011.
6. A.B. Degtyarev and A.M. Reed. Synoptic and short-term modeling of ocean waves. In *Proceedings of 29th Symposium on Naval Hydrodynamics*, 2012.
7. Kazushige Goto and Robert Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.
8. Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.
9. Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.
10. Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.
11. Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
12. Laércio L Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, and Jean-François Méhaut. Charm++ on numa platforms: the impact of smp optimizations and a numa-aware load balancer. In *4th workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing. Urbana, IL, USA*, 2010.
13. Steve Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.
14. Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V Kale. Hierarchical load balancing for charm++ applications on large supercomputers. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 436–444. IEEE, 2010.