Balancing CPU Load for Irregular MPI Applications

Jörg KELLER [a], Mudassar MAJEED [b] and Christoph W. KESSLER [b]
[a] *FernUniversität, 58084 Hagen, Germany*
[b] *Linköpings Universitet, 58183 Linköping, Sweden*

Abstract. MPI applications typically are designed to be run on a parallel machine with one process per core. If processes exhibit different computational load, either the code must be rewritten for load balancing, with negative side-effects on readability and maintainability, or the one-process-per-core philosophy leads to a low utilization of many processor cores. If several processes are mapped per core to increase CPU utilization, the load might still be unevenly distributed among the cores if the mapping is unaware of the process characteristics.

Therefore, similarly to the MPI_Graph_create() function where the program gives hints on communication patterns so that MPI processes can be placed favorably, we propose a MPI_Load_create() function where the program supplies information on the relative loads of the MPI processes, such that processes can be favorably grouped and mapped onto processor cores. In order to account for scalability and restricted knowledge of individual MPI processes, we also propose an extension MPI_Dist_load_create() similar to MPI_Dist_graph_create(), where each individual MPI process only knows the loads of a subset of the MPI processes.

We detail how to implement both variants on top of MPI, and provide experimental performance results both for synthetic and numeric example applications. The results indicate that load balancing is favorable in both cases.

Keywords. Load balancing, MPI applications

Introduction

The Message Passing Interface (MPI) [9] is a widely used standard for programming massively parallel machines. Typically, an MPI application assumes that each of its processes incurs a similar load on the core it runs. A second assumption, resulting from the first one, is that each MPI process should run on a core of its own for maximum performance by giving the application maximum control over the resources. There are tools such as SCALASCA [6] that aid users to detect imbalances that manifest e.g. as differing waiting times of processes at a barrier (or other collective operation), so that users can change their code to re-establish the assumptions above. There are even efforts to automate such change for applications with domain decomposition by adjusting the borders of the decomposition [2].

However, changing an existing MPI application to balance computational load often involves a lot of work and leads to unnatural and complicated code that is hard to debug and maintain. If the different load levels persist during a whole program run, and could be communicated from the application to the MPI system, then several lighter processes could be mapped onto one core without negative effect on the runtime[1], and the application could — without costly changes — either run with the same performance on a smaller machine, or a larger application could be run on the current machine. This aspect gains some importance with the introduction of multicore processors that enable compact parallel machines of medium size. For example, if half of the MPI processes have computational load 1 each, and half of the MPI processes have computational load 0.1 each, then the application can be run on $p = 0.55 \cdot n$ cores with the same performance, if 10 lighter processes get packed on one core.

An example of the application giving hints about its behavior to the underlying MPI system is the function `MPI_Graph_create()` [9], where the processes specify a communication graph with relative edge weights. The MPI system then has the chance to reorder the ranks of the MPI processes such that processes that frequently communicate larger volumes of data are placed closer together in a machine with a hierarchical communication network. In OpenMP [3], the pragmas represent programmer knowledge to the compiler, yet cannot incorporate information only available at runtime, such as problem size or characteristics of input data that influence computational load among processes. Multicore operating systems such as Linux migrate processes in the face of imbalances, yet MPI applications normally span multiple operating system instances. In [1], [4] and [7], dynamic load balancing in MPI applications by process migration and/or usage of threads is presented. This resembles techniques from dynamic scheduling and process migration in grid computing, and deeply cuts into the MPI implementation, while our goal is to achieve load balancing with minor interference of the underlying system. In [5], load balancing for divide-and-conquer MPI applications was presented, while we focus on applications with arbitrary communication patterns but stable computational loads. HeteroMPI [8] targets load balancing of MPI applications on heterogeneous machines by using performance models provided to the system. Our approach is useful also for homogeneous systems, which still present the mainstream in the medium range.

Therefore, we propose a function similar to `MPI_Graph_create()`, where the n MPI processes supply an array of n relative computational loads, and create a new communicator where the ranks of the processes are reordered such that CPU loads are balanced. For similarity, we name this function `MPI_Load_create()`. Despite the naming, it need not necessarily be integrated into the underlying MPI system, as it can be implemented with standard MPI functions, and thus can be seen on top of MPI. Reordering ranks of MPI processes for load-balancing is a typical task assignment problem and thus is NP-complete. As we solve it during runtime, we cannot afford expensive algorithms and thus use a simple heuristic.

Note that `MPI_Graph_create()` and our function both perform a reordering of ranks, but no process migration. However, the typical place to execute either function is directly after calling `MPI_Init()`, i.e. at program start, so that the MPI processes did not yet produce state such as data arrays or intermediate results that would have to be migrated. Hence, the rank reordering is sufficient. As thus, our function does not intend to provide fully dynamic load balancing at runtime, but ensures load balancing at program start. In contrast to static load-balancers that act at compile-time, the application at program start has some knowledge about the actual data set (for example by getting

[1]Note that for simplication we assume here that memory consumption of a process is proportional to its computational load. Otherwise, negative cache effects and additional paging might influence runtime.

core	0	···	7	8	···	15
ranks	0	···	7	8	···	15
	16	···	23	24	···	31
	32	···	39	40	···	47
	48	···	55	56	···	63

core	0	···	7	8	···	15
ranks	0	···	7	8	···	15
	16	···	23	24	···	31
	32	···	39	40	···	47
				48	···	55
				56	···	63

Figure 1. Initial round-robin rank orders for two process distributions

the data set size from the command line parameters), which can be used to provide appropriate computational weights. If an application would consist of several phases with different load behaviour, then a re-assignment is possible at runtime, however, the programmer would have to provide the data migration. However, this is also the case for `MPI_Graph_create()` and thus not considered here.

Obviously, to enable a load balancing as just described, an MPI application with n processes must be started on p cores with $n > p$, and typically n being an integral multiple of p, at least in the absence of dynamic MPI process creation.

In the following, we will explain how such a function can be implemented in an MPI system. We will also discuss an extension to provide scalability to large applications and systems. There it is neither feasible nor practical that each MPI process supplies an array of size n. In similarity to the MPI function `MPI_Dist_graph_create()`, where each MPI process only knows part of the communication graph, we propose a function `MPI_Dist_load_create()` where each MPI process only knows part of the array. In the extreme case, each MPI process may only know its own computational load. An experimental evaluation of a prototype implementation of our proposal indicates that it brings the performance benefits envisioned. Finally, we briefly discuss how implementation of `MPI_Load_create()` may be also beneficial to `MPI_Graph_create()`, i.e. how these two functions could be implemented and used in combination.

The remainder of this article is organized as follows. In Sect. 1, we present the mapping problem induced by the load balancing task, give a solution and explain how to implement it within MPI. In Sect. 2, we extend this solution to a distributed algorithm. We provide an experimental evaluation in Sect. 3, and give a conclusion and outlook in Sect. 4.

1. Load-balancing MPI applications

1.1. Ideally restricted scenario

We consider the scenario that we have started an MPI application with n processes on a parallel machine with p cores, where n is an integral multiple of p. We assume that in this case, $k = n/p$ MPI processes are mapped onto each core. The MPI process P_i, where $i = 0, \dots, n-1$, runs on core $i \bmod p$, as illustrated in Fig. 1. Each process P_i will have initialized MPI by a call to `MPI_Init()` and have got rank $r_i = i$. This scenario is very restricted, and will only hold for the communicator MPI_COMM_WORLD, if at all. Yet, this is only to start with. We will remove the restrictions one by one in the following paragraphs.

Each process has in its local memory an integer array $load$ of size n, where $load[i]$ contains the relative computational load of the MPI process with rank i. Thus, initially, a core j hosts processes with ranks $j, j+p, \ldots, j+(k-1)p$ and thus has load

$$l_j = \sum_{i<k} load[j + i \cdot p] \ .$$

Our goal is to permute the ranks of the processes by a permutation π such that the new core loads

$$l'_j = \sum_{i<k} load[\pi(j + i \cdot p)]$$

ensure that $\max_j l'_j \leq \max_j l_j$, i.e. that the computational load on each core is more balanced than before. This is a constrained variant of an optimization problem where it is required that exactly k processes be mapped to each core. In the unconstrained optimization problem, it might be necessary to adapt the number of processes per core.

Consider the following example, that we will use throughout the paper. We have $n = 64$ MPI processes, where those with ranks less than 20 have load 4, those with ranks between 20 and 39 have load 2, and the others have load 1. In total the processes provide load $144 = 20 \cdot 4 + 20 \cdot 2 + 24 \cdot 1$. If we run the processes on $p = 16$ cores, then the maximum load of a core must be at least $9 = 144/16$. This can be achieved if 10 cores host 3 processes each, with loads 4, 4, and 1, 4 cores host 5 processes each, with loads 2, 2, 2, 2, and 1, and 2 cores host 7 processes each, with loads 2, 2, 1, 1, 1, 1, and 1. If however each core runs exactly 4 processes, then the maximum load of a core is at least 10, because there must be cores hosting two processes with load 4 each, as there are more than $p = 16$ processes of that load. As those cores will host 4 processes, and the remaining two processes per core each have load at least 1, there will be cores with load at least 10. This example while still not very exotic, already puts some burden on balancers, because there is an integral optimal balancing, albeit a tricky one, and typical strategies for assignment lead to additional loads.

A simple heuristic to spread the loads is the following. Let us assume that element i of the array $load$ is a tuple $\langle ld_i, i \rangle$, where ld_i is the original content of $load[i]$. We sort the array in descending order with respect to the first component of the tuples. Then, if in the sorted array $load[i] = \langle ld_j, j \rangle$, we set $\pi(i) = j$. Thus, we distribute the loads round robin over the cores, heaviest loads first. This heuristic can be easily computed locally by each MPI process P_i, because it already possesses the array $load$. So it can extend the array, sort it, and lookup its new rank.

In terms of MPI, a call to `MPI_Load_create()` will compute a new communicator for the n processes, of the same size as the previous one, but the new ranks are the permuted ones. The function thus can even be implemented in C as a library to be linked to an application, creating the new communicator after the sorting by calling the existing function `MPI_Comm_split` with color 0 and key $\pi(i)$.

Getting back to our example, we would achieve maximum core load 11, because e.g. core 0 would host two processes with load 4, one with load 2, and one with load 1. Thus, the heuristic in this case is pretty close to the optimum. The resulting load distribution is depicted in Fig. 2.

core	0	$\cdots$	3	4	$\cdots$	7	8	$\cdots$	15
	4	$\cdots$	4	4	$\cdots$	4	4	$\cdots$	4
	4	$\cdots$	4	2	$\cdots$	2	2	$\cdots$	2
loads	2	$\cdots$	2	2	$\cdots$	2	1	$\cdots$	1
	1	$\cdots$	1	1	$\cdots$	1	1	$\cdots$	1

core	0	$\cdots$	3	4	$\cdots$	7	8	$\cdots$	15
	4	$\cdots$	4	4	$\cdots$	4	4	$\cdots$	4
	4	$\cdots$	4	2	$\cdots$	2	2	$\cdots$	2
loads	2	$\cdots$	2	2	$\cdots$	2	1	$\cdots$	1
							1	$\cdots$	1
							1	$\cdots$	1

Figure 2. Heuristic round-robin load distribution for two process distributions

1.2. Removing restrictions

The first restriction to be removed is the restriction that initially, process P_i has rank i. If the ranks are arbitrarily distributed, an appropriate order of the ranks can be achieved if each process knows on which core it runs. On a Linux-like operating system[2], a process can find this out by calling `sched_getcpu()`, a function in glibc since version 2.6.

The processes perform an `MPI_Allgather()` operation, where each process sends the index of the core it runs on. Each process thus has already learned the number p of cores in use. Then, a process running on core j computes the number s of processes with smaller rank and running on the same core. Then it assumes rank $j + s \cdot p$.

The second restriction to be removed is the requirement that an equal number of processes must run on each core. If this number is not equal, then we proceed as follows. First, after the `MPI_Allgather()` operation, we renumber the cores such that the ones with fewer processes get lower indices. Then, the ranks are assigned round-robin again. If cores have exceeded their process count, they are ignored in the next round. This is easy as those will always be the first ones, because the cores are renumbered by process count in ascending order. For example, if in our example from above, 8 cores will run 3 MPI processes each, and 8 cores will run 5 MPI processes each (for a total of 64 MPI processes), then the 8 cores with 3 processes will get indices 0 to 7, and the others 8 to 15. The processes on core 0 will be assigned ranks 0, 16, 32, the processes on core 1 will be assigned ranks 1, 17, 33, the first three processes on core 8 will be assigned ranks 8, 24, 40, and the first three processes on core 15 will be assigned ranks 15, 31, 47. Rank 48 will then be assigned to the fourth process on core 8, because after the third round, cores 0 to 7 have exceeded their process count. This ordering is illustrated in Fig. 1. The processes on core 0 will have highest load $10 = 4 + 4 + 2$.

The third restriction, that n should be an integral multiple of p, is obviously removed with the second restriction. If n is not an integral multiple of p, then simply the number of processes per core varies, which is allowed since the second restriction is removed.

[2]In MPI systems spanning multiple operating system instances, a tuple consisting of a hash of the machine name and the core id can be used.

1.3. Further balancing opportunities

While removing the second restriction widens the applicability of our proposal, the proposed function still is restricted to the distribution of processes over cores as given. A workaround could be to start the MPI application with $n + m \cdot p$ processes instead of n processes, i.e. with $k + m$ processes per core instead of k. The additional m processes per core could sit idle as long as they are not needed, and by being swapped out would neither consume much processing power nor much memory. They should however participate in the `MPI_Load_create()` with load 0. Thus, the load balancing heuristic has a better chance to compute a load distribution close to the optimum, because these extra processes are useful in two situations: when the resulting distribution would need a varying number of processes per core to provide good balance, and when the current distribution of (active) processes per core is not very suited to the task loads, so that it should be evened out. The extra processes could even be shared by several sub-communicators. In principle, this is a form of soft process migration, i.e. we reorder ranks among busy and idle processes, so that the number of busy processes per core may vary while the number of processes per core remains constant.

In our example, if we would provide $m = 3$ extra processes per core, and would employ an optimal load balancer, we could achieve the optimal maximum load of 9, as given above. We admit that the round-robin heuristic is not very well suited for the case of extra processes. If we provide $m = 1$ extra process per core, and use the round-robin heuristic in the case of 8 cores with 3 active MPI processes each and 8 cores with 5 active MPI processes each, then we get the same distribution as in the case of 4 MPI processes per core, and a maximum core load of 11 as explained above, while the heuristic achieved a maximum core load of 10 without extra processes. For optimal load balancers however, usage of extra processes will not lead to an anomaly as just observed.

2. Distributed load-balancing variant

The load-balancing algorithm of the previous section uses $O(n + p)$ space per MPI process. While this is no problem for small and medium installations and applications, it prevents scalability for large machines, i.e. $n, p \geq 10^4$. Therefore, in similarity to the function `MPI_Dist_graph_create()`, we also want to provide a function `MPI_Dist_load_create()`. The purpose of this function is, as in `MPI_Load_create()`, to provide a new communicator with balanced computational load, but in the distributed function each process only provides a list of tuples consisting of rank and load. In the extreme case, each process only provides its own load. Obviously, for small n and p, `MPI_Dist_load_create()` could be implemented by completing the load information on each process via an `MPI_Allgather()`, and then calling `MPI_Load_create()`.

For the distributed algorithm, we work in $k = n/p$ rounds. In the first round, we perform some precomputations and deal with heavy loads. In the remaining rounds, we assign the remaining loads. Our goal again is to spread the heavy loads over the cores. To do this, the processes first execute two `MPI_Allreduce()` operations on their load values, one with the MAX and one with the SUM operator. Thus each process knows the maximum load l_{max} of a process, and, by dividing the sum through the process

core	0	1	2	3	4	$\cdots$	7	8	$\cdots$	15
	4	4	4	4	4	$\cdots$	4	4	$\cdots$	4
	2	2	4	4	2	$\cdots$	2	2	$\cdots$	2
loads	4	1	2	1	2	$\cdots$	2	1	$\cdots$	1
	1	4	1	2	1	$\cdots$	1	1	$\cdots$	1

Figure 3. Round-robin load distribution computed by the distributed heuristic

count n, the average process load l_a. Now each process prepares a vector v of length t. Element i of this vector, where $0 \leq i \leq t-1$, is set to 1 if the process load is larger than $2^{t/2-i} \cdot l_a$, and to 0 otherwise. This means that for a process with load exactly l_a, the first $t/2+1$ elements will be zero, and the remaining elements will be one. Then the processes perform an `MPI_Allreduce()` with the SUM operator, receive a sum vector sv, and thus learn the number of processes with loads larger than $2^{t/2} \cdot l_a$, $2^{t/2-1} \cdot l_a$,..., $2^{-t/2+1} \cdot l_a$. The parameter t is a tuning parameter of our algorithm that allows to trade off expected load balancing quality for time and space requirements of the algorithm. In most situations, we believe $t = 16$ to be sufficient. If it turns out during runtime that it is not, then the processes could double t. If the scaling is too coarse, i.e. if all processes belong to one or two categories, the scaling might also be adapted.

In our example, we get maximum load $l_{max} = 4$ and average load $l_a = 144/64 = 2.25$. Processes with loads 4, 2, and 1, will prepare vectors $(0,0,0,1,1,1)$, $(0,0,0,0,1,1)$ and $(0,0,0,0,0,1)$, respectively, if $t = 6$. The processes receive a sum vector $sv = (0,0,0,20,40,64)$.

Each process now seeks for the last element in the sum vector with values less or equal to p, i.e. for the maximum i with $sv(i) \leq p$. All processes perform an `MPI_Scan` operation with the ADD operator. The ones with $v(i) = 0$ and $v(i+1) = 1$ provide a 1, the other processes provide a 0. The processes with $v(i) = 1$, and the processes that provided a 1 in the scan and that got a return value of at most $p - sv(i)$ are the heavy ones and are distributed over the cores. This is achieved if the $sv(i)$ processes with $v(i) = 1$ send their ranks and loads to the processes with ranks 0 to $sv(i) - 1$, and each process with $v(i) = 0,\ v(i+1) = 1$ and return value $j \leq p - sv(i)$ sending its rank and its load to the process with rank $j - 1 + sv(i)$. Here we again assume that the ranks are ordered round-robin over the cores. In our example, $i = 2$, $sv(i) = 0$ and $sv(i+1) = 20 > p$. Thus, the processes contesting to be heavy ones all have load 4.

In the second round each process seeks for the maximum i with $sv(i) \leq 2p$. In the `MPI_Scan` operation, the ones with $v(i+1) = 1$, that are not yet assigned, provide a 1. Thus, after k rounds, the loads are distributed. The round-robin is not as strict as in the heuristic of the previous section. While each load assigned in a previous round is heavier than any load of the next round, the loads may have an arbitrary order within the round.

Finally, the processes perform a `MPI_Comm_split()` operation as in the algorithm of the previous section.

In our example, in the second round $i = 3$, $sv(i) = 20$ and $sv(i+1) = 40$. Hence, contestors here are 20 processes with load 2, and 4 processes with load 4, as 16 processes with load 4 are already assigned. 16 of those 24 processes will be assigned, but it is not guaranteed that all processes with load 4 are among them. A possible load distribution resulting from the distributed algorithm is depicted in Fig. 3.

Note that one may also use an arbitrary assignment for the last round or rounds, in order to save computation time, without sacrificing load balance too much.

Table 1. Runtimes for synthetic benchmark program

Scenario	Optimal	Heuristic	Pseudo-random	Worst case
4 processes/core	18.6s	20.7s	22.0s	29.3s
3/5 processes/core	18.3s	18.3s	20.5s	38.6s

3. Experimental Evaluation

We have implemented the `MPI_Load_create()` and `MPI_Dist_load_create` functions as described in Sect. 1 with the help of available MPI functions. We use system dependent functions only to get IP addresses and core IDs. Hence, our implementation is highly portable. The implementation of the two functions comprises about 50 and 100 lines of C code, respectively. In order to validate our load balancing heuristic, we executed experiments representing some different load distributions from Sect. 1.

We performed experiments on a machine M1 with 4 quad-core AMD processors (2.2 GHz) with 32 GByte main memory, running under Linux, and on a machine M2 with 805 nodes, each with two 2.33 GHz Xeon quad core chips and at least 16 GiB RAM, running under CentOS5. The nodes are interconnected by Infiniband.

In our first experiment, we measured the time to call `MPI_Load_create()`. With 64 MPI processes, the call needs between 5 and 8 milliseconds on machine M1 and about 2.5 milliseconds on machine M2 with 8 cores on 1 node. With 1,024 MPI processes on machine M2 with 512 cores (64 nodes with 8 cores each), this time increases to only 36 milliseconds, i.e. a linear increase. As an application typically will call this function only once, and as it is typically used for applications running longer than a second, we consider this overhead negligible.

In our second and third experiments, we used a synthetic benchmark with $n = 64$ processes on $p = 16$ cores, and assigned with ranks as given below. After assigning the ranks, each process performed a loop where it added integers, the length of the loop being proportional to the load the process shall have. After the loop, the processes perform a barrier synchronization. This is repeated 10 times to exclude cache miss effects or other disturbances. For each experiment, we measured the wall-clock runtime, as no other user jobs were running on the machine, and the operating system load was considered negligible. The loads of our application are as in the example of Sect. 1: 20 processes with load 4, 20 processes with load 2, and 24 processes with load 1.

For the second experiment on machine M1, we pinned 4 processes to each core, and used the round-robin ranking. As a worst case scenario, we used a load profile where all processes on a core had the same load, so that e.g. core 0 has load $16 = 4 \cdot 4$, and there are cores with load $4 = 4 \cdot 1$. Then, as an average scenario, we assigned the loads pseudo-randomly to the processes, and obtained a maximum core load of 12. As an optimal scenario, we applied a distribution with maximum core load 10, and for the heuristic scenario, we applied the round-robin load assignment of the previous section with maximum core load of 11, which resulted from the call to `MPI_Load_create()`. The runtimes are depicted in Table 1. In the runtime measurements, we excluded the initialization, because it is similar in all cases and seems negligible for runtimes of more than 10 seconds. We see that the runtime of the heuristic load balancing is better than the pseudo-random load distribution which has 6% longer runtime.

For the third experiment on machine M1, we had 8 cores with 3 processes each, and 8 cores with 5 processes each. In this case, the `MPI_Load_create()` heuristic

Table 2. Runtime ratio unbalanced to balanced for LU-multiply benchmark

Scenario	1	2	3	4
matrix dim m	1,024	1,024	4,096	9,216
no. MPI processes n	64	256	256	256
no. cores	8	32	32	32
no. nodes	1	4	4	4
speedup `MPI_Load_create`	1.198	1.073	1.210	1.269
speedup `MPI_Dist_load_create`	1.059	0.971	1.082	1.154

achieves an optimal maximum core load of 10, although with a different distribution. For the worst case scenario, we used a distribution where core 15 had 5 processes of load 4 each, for a maximum load of 20. The average scenario again had a pseudo-random load distribution. The runtimes are also depicted in Table 1. We see a similar behavior as in the case of an equal distribution of processes over cores, only that the runtime in the average scenario is 11% longer than with the heuristic load distribution.

For a fourth experiment on machine M2, we implemented the multiplication of a lower triangular $m \times m$ matrix L with an upper triangular $m \times m$ matrix U by a variant of Cannon's algorithm. We pin n/p MPI processes to each core[3]. We have a square number $n = s^2$ of MPI processes, each computing a block of size $m/s \times m/s$ of the result matrix. Blocks from L or U that only contain zeroes are not used, so that the load for MPI process (i, j) is about $1 + \min(i, j)$. Table 2 presents the speedup, i.e. the ratio of runtimes without and with usage of `MPI_Load_create` and `MPI_Dist_load_create` for different matrix sizes and core counts. The ratio for `MPI_Load_create` closely matches the theoretical load ratio, if the number of processes per core is large enough (for Scenario 2, it seems too small). We also ran the experiment for $m = 4,096$ with $n = 64$ processes on 64 cores, i.e. with one process per core, and achieved the same runtime as scenario 3 with 32 cores.

The maximum time to execute `MPI_Load_create` is 11 milliseconds in scenario 4, for `MPI_Dist_load_create` it is about 90 milliseconds. Thus, the centralized variant's overhead is negligible because it is less than one percent even in scenario 2 (less than 10^{-5} in the others). The overhead is notably higher for `MPI_Dist_load_create`, which is the reason for the speedup less than 1 in scenario 2. A possible reason might be that the number of rounds is not well adapted to the load situation. Also, the runtime advantage of `MPI_Dist_load_create` is smaller than for `MPI_Load_create`, as a consequence of the load ordering not being strictly round robin in that case.

4. Conclusion

We have proposed a load-balancing function `MPI_Load_create()` where MPI applications with uneven computational load per process can give hints to balance this load, similar to what the `MPI_Graph_create()` function does with respect to communication loads. The only prerequisite is the presence of multiple MPI processes

[3]We also tested the runtime without the load balancing and without pinning processes to cores. Surprisingly, the runtime gets worse in scenarios 2 to 4.

per core, which is a necessity for any load balancing of this kind. We implemented this function as a library without change of the underlying MPI system. An advantage of our load-balancing function is that it does not require code changes in existing MPI applications with known imbalance in computational load, besides the call to `MPI_Load_create()`. We have also presented how to implement a distributed variant of our MPI function, similar to `MPI_Dist_graph_create()`, for large MPI applications and installations. We have provided an experimental evaluation which indicates that usage of our function indeed provides runtime advantages[4].

In the future, we plan to optimize the distributed variant of our load balancing scheme, and to provide programming support for data migration in applications with several phases. We also would like to investigate how the functions `MPI_Load_create()` and `MPI_Graph_create()` could profit from each other. Obviously, if both optimizations are performed together, there is a chance for profit by mapping pairs of processes with heavy communication load on the same core. Thus, one has to solve a multi-criterion optimization problem. In the distributed variant of our algorithm, one might be able to favorably combine both criteria, because after the first round, one may choose the ranks for each core according to the communication graph.

Acknowledgements

This work was partly supported by the Swedish e-Science Research Center (SeRC) project OpCoReS, and by a PhD fellowship of Higher Education Commission (HEC) Pakistan. We thank W. Schiffmann, P. Cichowski, U. Hönig and B. Wesarg for help with first tests on machine M1, and J. Träff for introducing us to MPI optimizations.

References

[1] M. Bhandarkar, L.V. Kale, E. de Sturler, and J. Hoeflinger, Adaptive load balancing for MPI Programs, in *Proc. Int.l Conference in Computational Science*, pp. 108–117, 2001.

[2] D. Böhme, M.-A. Hermanns, M. Geimer, and F. Wolf, Performance simulation of non-blocking communication in message-passing applications, in *Proc. 2nd Workshop on Productivity and Performance (PROPER 2009)*, 2009.

[3] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP — Portable Shared Memory Parallel Programming*, MIT Press, 2007.

[4] J. Corbalán, A. Duran, and J. Labarta, Dynamic load balancing of MPI+OpenMP applications, in *Proc. Int.l Conference on Parallel Processing*, pp. 195–202, 2004.

[5] M. Eriksson, C. Kessler, and M. Chalabine, Load balancing of irregular parallel divide-and-conquer algorithms in group-SPMD programming environments, in *Proc. 8th Workshop on Parallel Systems and Algorithms (PASA 2006)*, 2006.

[6] M. Geimer, F. Wolf, B.J.N. Wylie, and B. Mohr, Scalable parallel trace-based performance analysis, in *Proc. 13th European PVM/MPI Conference*, pp. 303–312, 2006.

[7] L. V. Kale, M. Bhandarkar, and R. Brunner, Run-time support for adaptive load balancing, in *Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP)*, pp. 1152–1159, 2000.

[8] A. Lastovetsky and R. Reddy, HeteroMPI: Towards a message-passing library for heterogeneous networks of computers, *Journal of Parallel and Distributed Computing* **66** (2006), 197–220.

[9] MPI Forum, *MPI: A Message-Passing Interface Standard Version 2.2*, 2009.

[4]So far we only found one large configuration where load balancing provides no advantage, possibly because the majority of time is spent with communication, but this will require further analysis in the future.