Automation of Determination of Optimal Intra-Compute Node Parallelism

Antonio Gómez-Iglesias Texas Advanced Computing Center
The University of Texas at Austin
Austin, Texas
Email: agomez@tacc.utexas.edu
James C. Browne Department of Computer Science
The University of Texas at Austin
Austin, Texas
Email: browne@cs.utexas.edu

Abstract

Maximizing the productivity of modern multicore and manycore chips requires optimizing parallelism at the compute node level. This is, however, a complex multi-step process. It is an iterative method requiring determining optimal degrees of parallel scalability and optimizing memory access behavior. Further, there are multiple cases to be considered, programs which use only MPI or OpenMP and hybrid (MPI +OpenMP) programs. This paper presents a set of three coordinated workflows for determining the optimal parallelism at the program level for MPI programs and at the loop level for hybrid (MPI+OpenMP) cases. The paper also details mostly automated implementations of these workflows using the PerfExpert infrastructure. Finally the paper presents case studies demonstrating both the applicability and the effectiveness of optimizing parallelism at the compute node level. The results shown in the paper will provide valuable information to further advance in the full automation of the workflows. The software implementing the parallelism scalability optimization is open source and available for download.

I. Background

A majority of the scientific and engineering applications executed on high performance computer clusters implement intra-node parallelism through MPI. While many if not most MPI libraries incorporate low overhead modes of communication among tasks executing on a single compute node, all the loops will normally have the same degree of parallelism. It is often the case however, that the optimal degree of parallelism may vary among the loops of the application. Therefore, customizing the level of parallelism of each loop may provide performance benefits. Applications that incorporate loop level parallelism using a threading package such as OpenMP are referred to as hybrid programs. Because threading at the loop level adds overhead there is a need to be able to estimate the potential benefit from adding loop level parallelism. Also, due to this overhead it may not always be beneficial to add loop level parallelization. This forces us to introduce a threshold that indicates when adding this parallelization may be

valuable. There is, thus, a need to be able to estimate the potential benefit from adding loop level parallelism. The degree to which memory access limits performance gain from parallelism may vary from loop to loop. Typically the main limiting factor in the degree of parallelism in a loop or function is memory access.

PerfExpert allows users to efficiently profile sequential and parallel applications in a variety of platforms. In this paper, we are going to focus our efforts on the Intel Sandy Bridge processor. This is the processor that can be found on the Stampede cluster, which is used for our experiments. Each node in Stampede is a dual socket eight-core Sandy-Bridge E5-2680 server node with 32 GB of memory. Each socket has 16 GB of local memory and a level 3 cache shared among the 8 cores in the socket of 20 MB. L2 size is 256 KB while L1 is 64KB. While the compute nodes of Stampede incorporate manycore Knights Corner chips, in this paper we demonstrate the workflows with application only to the multicore (CPU) chips of the Stampede compute nodes. The memory access analyses provided by PerfExpert can be used to determine for MPI only programs the potential performance benefit from adding separate parallelization of loops via OpenMP. The results presented in this paper combined with previous experiments performed on a diverse set of applications allow us to empirically create a set of rules that will a priori determine the theoretical highest degree of parallelism that a given loop or function can achieve on a specific platform.

Modern multicore and many core chips may have dozens of cores, sometimes incorporating multithreading on each core. Each core has a complex memory hierarchy embedded in additional layers of memory hierarchy at the chip level. Performance gains from parallelism are often determined or limited by memory access. Determining the optimal degree of parallelism (which we define as maximal productivity for the application) is a very complex task requiring in-depth knowledge of architectures and compilers.

There is therefore, a need for tool support to enable application developers and users to easily determine the optimal levels of parallelism for pure MPI programs, the extent of performance gains from adding OpenMP threading and the optimal levels of parallelism for hybrid programs.

This paper presents a set of workflows and a tool chain that automates a large fraction of the workflows for determination of optimal parallelism at the compute node level. These capabilities for optimizing parallelism are implemented in the modular infrastructure for implementing performance optimizations provided by the PerfExpert system [1]–[3]. Earlier versions of PerfExpert automated most of the steps in optimizing memory access at the chip and node level. This paper reports an extension of PerfExpert to optimize loop scalability. This paper utilizes the PerfExpert infrastructure to optimize parallelism. Additionally the analyses of memory access provided by PerfExpert are essential to enhance parallel scalability and can be used to determine for MPI only programs the potential performance benefit from adding separate parallelization of loops via OpenMP.

The main contributions of this paper can be summarized as follows:

- A set of three workflows for optimizing parallelism: (i) one for optimizing parallelism for pure MPI programs, (ii) one for estimating the benefit from adding OpenMP threading at the loop level to a pure MPI program, (iii) and one for determining the optimal parallelism of a hybrid program (MPI + OpenMP).
- An implementation of the workflows which almost completely automates determination of optimal parallelism at the compute node level as a tool.

- Demonstration of the applicability of the implemented set of workflows which shows that substantial improvement in application productivity can be obtained by use of the new tool.

The rest of the paper is organized as follows: Section II describes the workflows. Section III sketches how the PerfExpert infrastructure can be used to implement the determination of optimal scalability as well as the current level of automation of the different workflows in PerfExpert. Section IV gives a case study whereas Section V summarizes what has been presented and sketches further developments. Finally, Section VI presents the related work.

II. Workflows

In this section we introduce three different workflows, one for MPI programs, one for hybrid programs and one to estimate the benefit of adding separate parallelization of each loop to an MPI program. The letters in parentheses at the end of each step, either **(PE)** or **(M)** are how the step is executed. **(PE)** means that the PerfExpert system executes the step automatically, while **(M)** means the step must be done manually.

a) ***Definition 1:*** Local Cycles per Instruction (LCPI) is the metric used by PerfExpert to assess the degree to which each resource in the memory hierarchy is being effectively used. Microbenchmarks that are executed when PerfExpert is installed are used to generate values for this metric. These values are used to determine the effectiveness of memory access. PerfExpert computes an LCPI value for each loop and function and decomposes the contributions at loop level from each element in the memory hierarchy.

b) ***Definition 2:*** The memory access saturation table (Table II) which is used in the workflow for estimation of potential benefit from adding loop specific parallelism loop. Increasing the number of threads executing a loop increases the LCPI for that loop. Memory access usually becomes a bottleneck in the systems that we have run this analysis when LCPI is larger than 0.5. For serial codes, a larger LCPI might not represent an issue, but it quickly becomes a limiting factor for parallel applications.

A. *Optimal Productivity for Loops in MPI Programs*

The workflow for estimation of the benefit of adding loop specific parallelism also uses a threshold for determining when adding loop specific parallelism will be beneficial. This threshold is determined empirically for each execution environment.

Note that the optimal level of parallelism usually varies with inputs, particularly those that affect the amount of data which is processed. This workflow may need to be run separately for inputs which generate significant changes in the amount of data to be processed.

Therefore the first step for pure MPI codes is to characterize the performance of that MPI implementation using PerfExpert and to optimize the memory access behavior for that original implementation. Once the memory behavior has been defined, changes in the code might need to be introduced to effectively improve the usage of the memory. Those changes need to be reevaluated to measure their effectiveness. The loop is repeated until no further improvement in performance of the code. Therefore, the steps in the workflow for MPI programs (MPI Workflow) are:

1.1 If there are multiple chips in the compute node, check intra-node load balance. **(PE)**

1.2 Rebalance workload if necessary. **(M)**

1.3 Determine the optimal level of parallelism (number of tasks per compute node) for the application in its original state by varying the number of MPI tasks around the original number of tasks. Call this number of tasks T_0. **(PE)**

1.4 Optimize memory access behavior for T_0 for each loop that takes significant execution time. The process used by PerfExpert for optimizing memory access at the loop level is reported in several papers [3]. **(PE) (M)**

1.5 Rerun the scalability analysis (Step **1.3**) to see if the memory access optimization has improved the scalability of the application. **(PE)**

1.6 If the optimal number of tasks, T_0, has changed more than a threshold amount between Steps **1.3** and **1.5**, go to Step **1.4**.

1.7 Save T_0 and the memory access parameters for loops that take significant execution time for use in the workflow for estimating the benefit from separate parallelization of each important loop.

B. Estimating Potential Productivity Gain from Loop Local Parallelism

This workflow tries to estimate, for each loop or function that presents memory access bottlenecks, the benefit of increasing the local level of parallelism. This can be later on achieved by introducing OpenMP at the loop level to provide an specific level of parallelism for each specific loop or function. This workflow focuses on those loops and functions identified as key limiting sections of the code with the MPI workflow.

The steps in the workflow for estimating the potential benefit of loop level parallelism (Estimation Workflow) are:

2.1 After completing the MPI workflow, examine the LCPIs for each loop which were saved in the last step. **(M)**

2.2 Then, for each important loop, use the table relating data access LCPI for one task or thread to the number of task/threads which will lead to memory access bottlenecking in a given execution environment to estimate the number of tasks, T_s, necessary to generate saturation of the memory access. Linear extrapolation is sufficient since T_s will be re-optimized for each loop in the next workflow.

2.3 For each of those loops for which $\mid T_s\text{-}T_0 \mid$ is greater than a threshold value, loop level parallelization should be beneficial. The T_s for a loop is a reasonable initial value for the number of threads in an OpenMP declaration for the loop. If there are no loops for which $\mid T_s\text{-}T_0 \mid$ is greater than the threshold value, then there will usually be little benefit from adding threading to the MPI code.

C. Optimal Loop Local Parallelism/Productivity

The final workflow or use case consists of the application of the estimated parallelism to each of the individual loops or functions that were identified in the MPI Workflow and that are worth changing according to the Estimation Workflow. As aforementioned, we consider the use of OpenMP to apply this degree of parallelism at the loop level. There are two different cases of this workflow (Hybrid Workflow):

(a) The original application was already hybrid (MPI+OpenMP) and the goal is to optimize the parallelism for each important loop in the original code by using specific OpenMP constructs that indicate the number of threads to be used on each case.

(b) The application was only parallelized with MPI and the original application has been processed with the previous workflows before applying this workflow.

If (a) holds then the first four steps in this workflow are not needed. If (b) holds then, before beginning this workflow, the user or application developer must have added an appropriate set of OpenMP declarations. This will include the specification of shared and private data to each important loop and the use of the estimates for optimal parallelism for each of those loops developed in the Estimation Workflow.

It is most often the case that optimal or near optimal productivity will be obtained using one or two MPI tasks per chip and varying the degree of threaded parallelism for each loop.

The steps in the Hybrid Workflow are:

3.1 If there are multiple chips in the compute node, check intra-node load balance. **(PE)**

3.2 Rebalance workload if necessary. **(M)**

3.3 Determine the optimal level of task parallelism for the application in its original state. **(PE)**

3.4 Identify the important loops in the application and optimize the memory access behavior for each important loop at the original degree of threading of the loops. **(PE)**

3.5 If case (a), determine the optimal degree of parallelism for each of the important loops by varying the degree of parallelism (tasks + threads) in the OpenMP threading specification. If case (b), choose one or two MPI per chip and determine the optimal degree of parallelism for each of the important loops by varying the degree of parallelism (tasks + threads) in the OpenMP threading specification. **(PE)**

3.6 Re-optimize the memory access behavior for each of the important loops at the optimal level of parallelism for that loop. **(PE)** **(M)**

3.7 Determine the optimal degree of parallelism for each of the memory access optimized important loops (as in Step **3.5**). If the change exceeds a threshold value in any loop, re-apply Step **3.6** to that loop. **(M)**

III. Background

A. PerfExpert Infrastructure

PerfExpert is an open source tool that allows users to easily detect the causes for any performance bottlenecks of an application. It diagnoses the causes at the core, socket and node level for each function or loop of the code. It can also automatically apply a set of transformations on the code to enhance its performance. Finally, it provides a performance analysis report as well as a set of recommendations that developers can introduce in the code in the case that PerfExpert has not been able to implement the optimizations it recommends.

The tool presents a modular design that aims at creating an easy to extend framework that may be adapted to new technologies. Different modules can be added to this framework, each one of them targeting different aspects of performance measurement and optimization. It currently provides modules for memory access characterization,

vectorization, load imbalance, recommendations based on other analyses, support for different tools to collect data from the chip, modification of the code, and automatic recompilation, among others.

B. Automation of the Workflows with PerfExpert

We will focus on two of the currently existing modules in PerfExpert. While not all the steps of the workflows are currently implemented, they can be fully automated by adding the results presented in this paper. The two modules that we use for these experiments are the LCPI and the imbalance modules. The LCPI module automatically measures the main characteristics regarding memory access that indicate the cost of running the different loops and functions of the code by using either HPCToolkit [4] or Intel VTune Analyzer. This cost considers the number of accesses to each memory level together with the latency of each one of those levels. The imbalance module uses the results measured by the LCPI module and detects any intra-node load imbalance among the processes or tasks that are used during the execution of the code.

The two modules will create two different reports:

1) LCPI report: it provides a set of valuable metrics for each loop or function of the code that runs longer than a given percentage of the overall execution time. In our case, we are focusing on the information regarding data accesses. This information is calculated as shown in Eq. 1. In the equation, *L1_Hit*, *L2_Hit* and *L3_Hit* represent the number of hits in L1, L2 and L3 cache; *L1_latency*, *L2_latency* and *L3_latency* represent the latency time for each one of the caches; *L3_miss* is the number of misses in the last level cache; *MEM_latency* is the latency of the main memory; and *INST* is the total number of instructions executed.
2) Imbalance report: it shows, for each process or thread used in the application, the fraction of the work performed by that process/thread. PerfExpert considers the number of instructions executed by each process/thread to calculate these values. The final report also presents the variability of each process/thread.

$$\begin{aligned} LCPI = &(L1_Hit \times L1_latency + \\ &L2_Hit \times L2_latency + \\ &L3_Hit \times L3_latency + \\ &L3_miss \times MEM_latency) / \\ &INST \end{aligned} \tag{1}$$

For the scalability analysis, we will focus on the value calculated with the Eq. 1.

We have already introduced for each workflow which steps are currently automated in PerfExpert and which ones are manually executed. For the first and third workflows, re-balancing the workload when necessary is not possible with PerfExpert since it might need large code or input changes. Something similar might be required for the Step **3.7** of the Hybrid Workflow, where it might be necessary to individually change the degree of parallelism of individual loops in the code. This is a step that we plan on automating in the future by automatically changing the code and rerunning the application.

IV. Case Study

For the case studies presented in this paper we have used the SPPARKS code [5], a kinetic Monte Carlo simulator for all three workflows, and three loops from the miniFE code from the Mantevo [6] benchmarks for an additional illustration of the workflows. The implementation of the SSPARKS code presents many similarities to the well known code LAMMPS [7]. SPPARKS presents a set of options that make it an ideal candidate for this type of analysis: it allows selection of different memory access patterns that will allow verification of the validity of our approach for the MPI Workflow without requiring changes in the code. It also consists of a set of well-defined kernel functions that are easy to identify. However, since it is a pure MPI code, the application of the Estimation Workflow will show that incorporation of OpenMP parallelization will be be productive for the main loop in SSPARKS, something that will require more changes in the code. The well-defined kernels facilitate the hybrid implementation.

A. *Optimal Productivity for Loops in MPI Programs*

SPPARKS is a code that implements different applications and models. In our case, we have chosen to run an Ising model, an on-lattice application (details on how to select these options can be found on the SPPARKS website[1]). We first tried SPPARKS with a memory access model that performs random accesses to memory. Also, elements on a very large array are accessed based on the indexes contained in a second array. This is an inefficient memory access pattern that normally limits the scalability of applications due to bottlenecks since the random accesses introduce stall cycles in the execution of codes that present this type of behavior. When running SPPARKS, we also found that there are cases that can present a large load imbalance. Even though the code implements a mechanism that tries to efficiently partition the problem space among the processes involved in the computation, some characteristics of the input might lead to scenarios that are impossible to balance. For example, it is common to find the results in Table I for two different input files. Both inputs specify the same number of particles, or problem size, but with a different spatial distribution. Indeed, in many cases it will not be possible to change the distribution of the particles when performing the simulations. However, after inspecting the code, it is clear that changing it to improve the load balance for come cases is a complex task that is outside of the scope of this paper. We will focus on datasets that present optimal load balance among tasks (Step **1.1**).

Fig. 1 shows the strong scalability of the solve component of SPPARKS (it does not include communication and I/O) for three different problem sizes. The code presents super speedup since the number of calculations does not decrease linearly with the number of particles when the same number of particles is divided among more processes. Considering the previous statements about memory access, this is somehow a surprising result. This figure can be misleading, as it seems to indicate that the code performs optimally. However, because of the algorithm that the code implements, when the original problem is divided into smaller subproblems, each MPI process effectively solves a much smaller problem than what the subproblem size seems to indicate. The number of calculations significantly decreases and that leads to this super speedup. When a weak scalability analysis is performed (see Fig. 2), it is

[1] http://spparks.sandia.gov/

Task	Case 1 %	Case 2 %
0	17.94	6.31
1	5.52	6.28
2	5.54	6.29
3	5.52	6.30
4	5.48	6.31
5	5.55	6.30
6	5.51	6.32
7	5.51	6.29
8	5.47	6.20
9	5.46	6.20
10	5.44	6.20
11	5.44	6.22
12	5.39	6.19
13	5.41	6.21
14	5.41	6.19
15	5.41	6.20

TABLE I
LOAD BALANCE ON MPI TASKS

possible to see that the code presents issues that limit its scalability. For small cases, since large portions of the problem fit into the different levels of cache, the code presents good scalability up to 5 cores, and not bad results up to 7 processes. With 8 or more tasks the scalability significantly decreases. This is even worse with larger problem sizes, where each core needs access to main memory. For medium size problems, the code presents optimal results with 2 tasks, and it could be used with 3-4 cores. After that point, the penalty for increasing the number of tasks is too big. And for large problem sizes, it is even more clear that 2 processes is the optimal configuration and that using more tasks introduces a big decline in the computation time. Therefore, after Step **1.3**, the value of T_0 is 5 for small problem sizes and 2 for the rest.

PerfExpert detected that the most time consuming part in the code is the `site_energy` function. The percentages of the time spent in this function with the previously presented cases for weak scalability are depicted in Fig. 3. This function normally takes more than 60% of the overall execution time.

Finally, the LCPI for the `site_energy` function is shown in Fig. 4. It is possible to see the relation between this figure and the one displaying the weak scalability of the code.

The next step in our workflow (Step **1.4**) is to optimize the memory access behavior for each loop or function. We chose SPPARKS because it allowed easy modification of the memory access pattern by simply selecting a different model in the input configuration. Instead of a random access, we now use a model that implements an effective usage of data locality. The results (Step **1.5**) are shown in Fig. 5 for the same problem sizes previously used. The LCPI in this case is always in the range [0.27, 0.29], and the percentage of the execution time spent in the `site_energy` function is now between 22% and 27%. While the weak scalability has significantly improved, the results indicate that it still slightly degrades for larger tasks counts. For small and medium problem sizes the

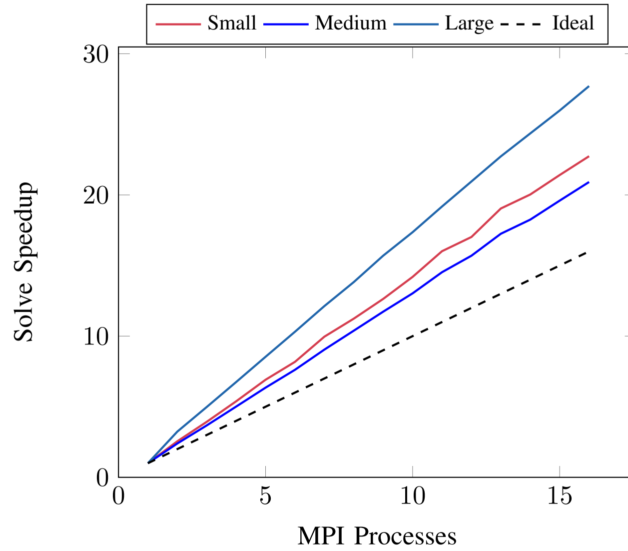

Fig. 1. Strong scalability of the solve function in SPPARKS for three different problem sizes and random memory access

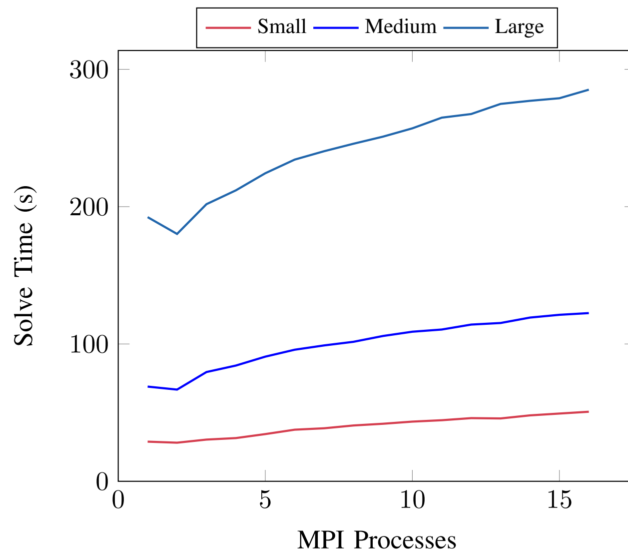

Fig. 2. Weak scalability of the solve function in SPPARKS for three different problem sizes and random memory access

value of T_0 still remains the same. However, for the case of large problems the value of T_0 has increased from 2 to 8, larger or equal than the threshold that we consider to be 6 for the Stampede compute node for this particular step of the workflow (Step **1.6**). This indicates that possible new optimizations can be introduced in the memory access pattern for large problems. However, because of the already very low LCPI calculated, and since it is not possible to find changes that can be easily implemented in the memory access pattern to improve an already excellent memory access pattern, we continue with the following workflows instead of iterating again. Thus, focusing on large problem sizes, after Step **1.7** the value of T_0 is 8.

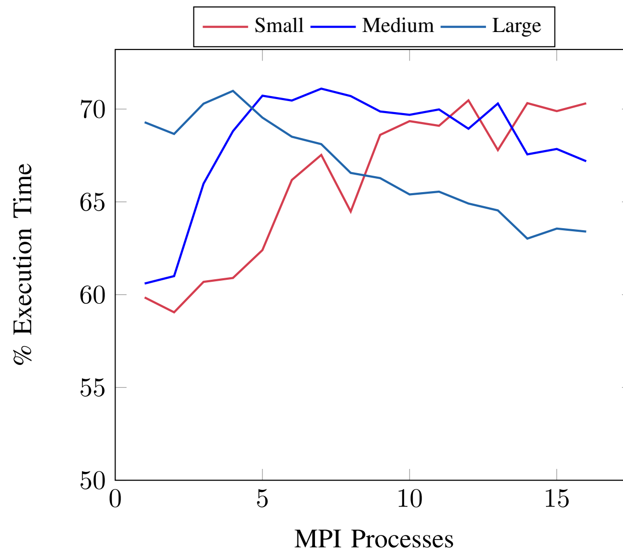

Fig. 3. Percentage of time spent in the `site_energy` function

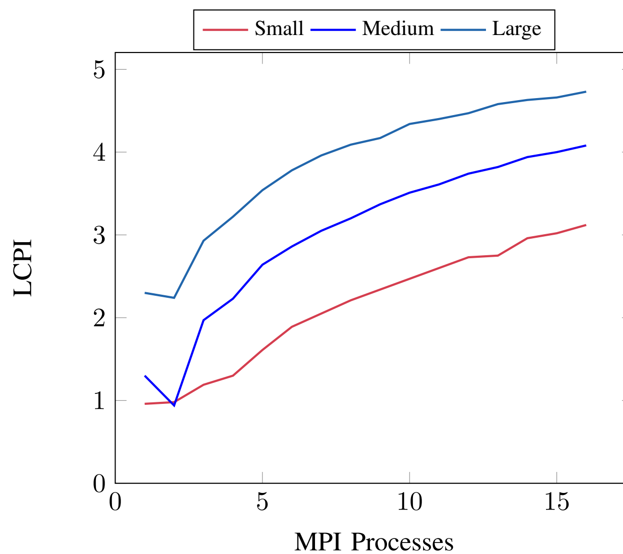

Fig. 4. LCPI of the `site_energy` function of SPPARKS for three different problem sizes

B. Estimating Potential Productivity Gain from Loop Local Parallelism

Table II relates small ranges of values for the LCPI metric to the scalability of a section of the code with this metric for the Intel Sandy Bridge processor and the Stampede compute node structure. The values in Table II include the overhead added by OpenMP. For small LCPI values the rate of change of parallel scalability is rapid but as memory saturation is reached, the rate of change becomes slower. Bear in mind that the estimates of scalability obtained from Table II will be refined in the MPI Workflow if necessary. However, the case studies we have done

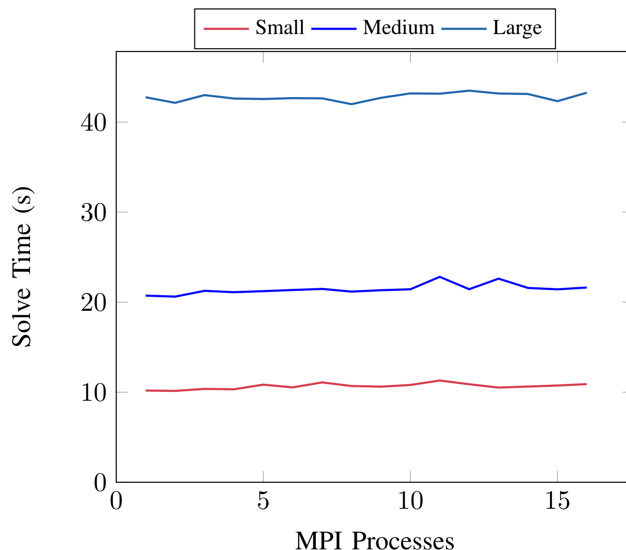

Fig. 5. Weak scalability of SPPARKS for three different problem sizes using an improved memory access pattern

show the estimates from Table II to be fairly accurate.

LCPI					Optimal Degree of Local Parallelism
0	≤	x	<	0.5	16
0.5	≤	x	<	0.6	14
0.6	≤	x	<	0.65	12
0.65	≤	x	<	0.7	10
0.7	≤	x	<	0.75	8
0.75	≤	x	<	0.8	6
0.8	≤	x	<	0.9	5
0.9	≤	x	<	1	4
1	≤	x	<	1.5	3
1.5	≤	x	<	2.5	2
2.5	≤	x			1

TABLE II
ESTIMATION OF THE OPTIMAL LEVEL OF LOCAL PARALLELISM BASED ON THE LCPI

With Table II, a single execution of a program gives an estimate of how well each loop or function will scale. It is necessary to consider the relative weight of each loop or function on the overall execution time. For SPPARKS, a single function, the `site_energy` function takes a large fraction of the execution time while no other single function presents a significant contribution to the overall execution time. Therefore for SSPARKS, the role of this workflow is to determine if threading is useful in the `site_energy` function. This function has LCPI values for large problem sizes showing that the function should effectively scale up to 16 processes (Step **2.2**). Therefore, following our workflow, $| T_s - T_0 |$ in this case gives a result of 8, since T_s = 16 for this section of the code, given

the LCPI in the range [0.27-0.29], and the value of T_0 found in the MPI workflow was 8. We have found that, when the difference between T_s and T_0 is larger or equal than the threshold value of 6 for the Stampede compute nodes, it is predicted to be beneficial to add loop level parallelism (Step **2.3**).

The SSPARKS example shows that adding threading parallelism to its single dominant function is beneficial. To complete the demonstration of the this workflow, we illustrate it with the main loops in miniFE in terms of LCPI and CPU which both have non-trivial execution time and can be parallelized with only local changes to the code. These three loops are: i) the `matrix-vector` product in SparseMatrix_functions; ii) the `dot` vector product in Vector_functions; iii) and the `daxpby` function also in Vector_functions. The loops have original LCPIs of 0.80, 2.71 and 2.11. After changing the code in the MPI Workflow, the LCPIs are 0.80, 1.37 and 2.31. The T_0 values for the loops are 4, 3 and 1, while the T_s values are 5, 4, and 1. Using Table II and the threshold value of 6 for the value of $\mid T_s\text{-}T_0 \mid$, there are no good candidates in this code for further exploration without large changes in the code. These changes are suggested by the high LCPI values that these functions present.

C. Optimal Loop Local Parallelism/Productivity

With the results from the previous workflows, we determined that the next step was to parallelize the `site_energy` function with OpenMP and optimize, if needed, the degree of threading recommended in the previous workflow. Since SPPARKS is a pure MPI code, this use case corresponds to case (b).

The function `site_energy` is a very short loop that simply iterates over the neighbors of a particle. Since the number of neighbors is small, parallelizing this function with OpenMP in fact increases the execution time due to the OpenMP overhead.

However, the function of interest is called many times from another function, once for each particle. Parallelizing this element allows a better distribution of the work and each loop gets a significant portion of work to perform. After parallelizing the function and running the code, it was clear that NUMA misses were a big penalty for the application, limiting again the scalability of the code. Therefore, another change during the initialization of the particles was introduced so that the data is allocated closer to the thread that will later require it.

While completely removing NUMA effects was not possible without large changes in the code, the simple strategy for parallelizing the code improved its performance. Only a few lines where added to the code to introduce the OpenMP pragmas. Since the changes affect mainly one section of the code, plus the initialization of the data, the number of threads used can be configured with environment variables. In many other cases, different loops will require different levels of parallelism, so the pragmas constructs will have to specify specific numbers of threads. If we had decided to continue with miniFE, the three loops would have presented different values (i.e., 5, 4 and 1).

After changing the code to achieve good weak scalability, we saw that theoretically, the maximum degree of parallelism that could be achieved for large problem sizes would be 16. We will show the results from the following configurations for our tests:

- 1 MPI task: 2 to 16 threads.
- 2 MPI tasks: 2 to 8 threads.

Larger numbers of MPI tasks were also checked but they did not improve the results achieved with 1 and 2 tasks.

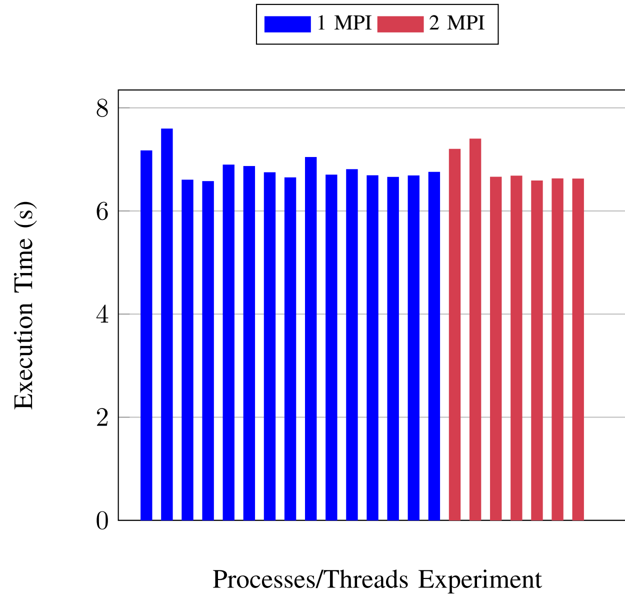

Fig. 6. Execution time for a small size problem

Table III shows the load balance for the case of 2 MPI tasks and 8 threads per tasks (Step **3.1**). The workload is well balanced between the two processes, each one of them running on its own socket.

Task	2 MPI - 8 Threads
0	49.37
1	50.63

TABLE III
LOAD BALANCE ON 2 MPI TASKS AND 8 THREADS PER TASK

We already know from the previous workflows the most important loop in the application. Figs. 6, 7 and 8 show the execution times for the previous combinations of MPI processes and threads for the three problem sizes presented throughout the paper.

The LCPI for the functions of interest in SPPARKS after changing the code remains below 0.3 for all the different problem sizes and configurations presented in this paper (Step **3.3**). This value suggests an optimal memory access pattern.

When the number of MPI processes remains constant, the execution time normally decreases. This in fact means that the weak scalability of the code has significantly increased. Even though it is not shown here, with more tasks, as the number of MPI ranks increases so does the execution time.

The value of T_0 for large problem sizes is now 16 and it is also very close to that value for medium size problems. This is the same value that we had previously calculated for T_s. This indicates the end of the loop since the ideal degree of parallelism has been achieved. For the case of small problem sizes, we see how the best results

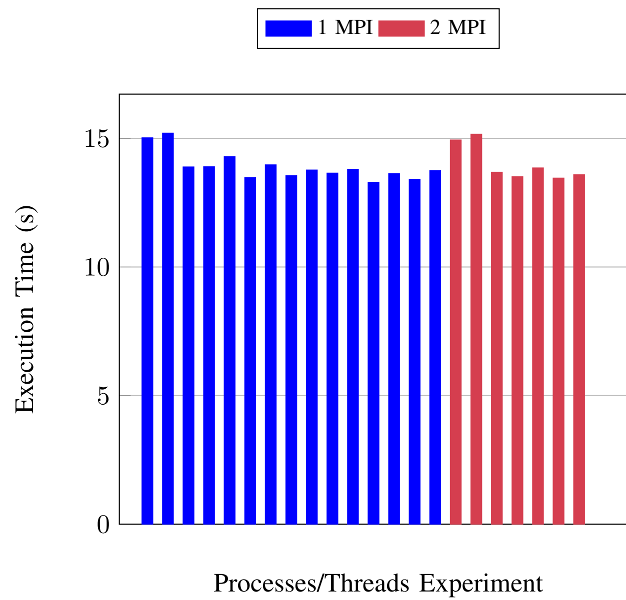

Fig. 7. Execution time for a medium size problem

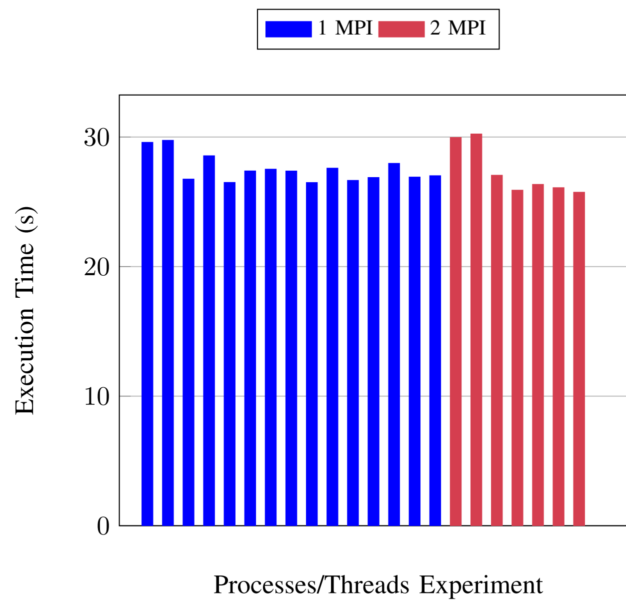

Fig. 8. Execution time for a large size problem

are achieved with 4 threads, but that there are no big differences between this case and 2 MPI tasks with 8 threads.

V. Conclusions

In this paper we have presented a collection of workflows that can be used to automatically determine the optimal level of parallelism at the loop level inside one node. We consider the workload of each process/thread as well as memory access patterns at the loop level. We have chosen an application that allows us to demonstrate the validity of our proposal without having to introduce a large number of changes, something that is outside of the scope of this paper.

We have shown that the LCPI metric, coupled with a threshold parameter, can reliably determine potential benefit of loop specific parallelism and accurately estimate the degree of parallelism for each loop. The information required for this process, a table relating LCPI to degree of parallelism leading to memory saturation and the threshold value for implementing loop specific parallelism can be derived with a small set of PerfExpert experiments.

We have shown how PerfExpert can be used to partly or fully automate these workflows. A modular framework has been introduced and some of the available tools included in PerfExpert used in these workflows have been also detailed.

A. Future Work

We plan on implementing most of the steps of the three workflows in PerfExpert. With the data presented, it is now possible to automatically implement a methodology that will only require of the intervention of a developer to implement some changes in the code.

For future work we consider fully automation of other aspects of modern processors. For example, vectorization needs to be considered since it is a more relevant factor in the performance of scientific applications with each new processor.

We plan on supporting new platforms, like the new Intel® Knights Landing. These platforms, with complex memory hierarchies, and large number of cores, both physical and logical, are ideal and challenging candidates for these workflows.

VI. Related Research

Determining optimal parallelism within a compute node of an HPC system which has multiple multicore or manycore chips for a complex application has been and is an extremely rich source of problems. The span of problems is rapidly increasing with the arrival of many core CPUs and GPUs. However, the problems are similar to those found in optimizing parallelism for the shared memory multi-processors of previous generations of parallel architectures.

Compute node level parallelism can be implemented in multiple ways: task level parallelism, usually with MPI, threading usually with OpenMP and hybrid where each task has multiple threads, usually OpenMP threads for the important loops in each task. There are three approaches to optimizing parallelism in application codes: (a) compiler optimization where a compiler uses source code level information to attempt parallelization of loops, (b)

optimizing parallelism based on runtime measurements and (c) autotuning where source code information is used to determine whether or not a loop is parallelizable and autotuning is used to select the optimal modes and parameters for the parallel code. Autotuning optimization of parallelism is thus a combination of (a) and (b) as is profile guided optimization of parallelization by a compiler but we are aware of only a few recent efforts using these approaches [8]. There is a large literature on parallelizing serial codes using several different methods. These papers generally do not attempt optimization.

All modern compilers provide some capabilities for parallelizing loops. This technology goes back to at least 1988 [9]. In order to successfully parallelize a loop, the compiler must the able to determine that the loop can safely be parallelized using dependence analysis and then determine whether or not parallelization is likely to be beneficial using a cost model. The applicability of parallelization based only on source code information is limited to cases where the dependencies can be determined and a reasonable estimate of the benefit of parallelization can be obtained. Similar limitations apply to autotuning to optimize parallelism although autotuning to estimate optimal degree of parallelism is practical or loops which are manually parallelized. In many cases the compiler will provide pragmas or directives by which the programmer can input information about the loops which will enable the compiler to parallelize the loop. The primary limitation of optimization based on runtime measurements is that the optimization is directly applicable only to the workload for which the measurements were taken.

This review of related research will focus on previous work where parallelism is optimized using information gathered with runtime information. Conceptual foundations for automating of optimizing of loop parallelism were given by [10]. Eigenmann and his collaborators [11], [12], [13] presented the implementation and application of the methodology. The ScaAnalyzer [14] identifies memory access bottlenecks which limit parallel scalability and associates them which data objects. It then optimizes access behaviors for these memory objects to enable greater parallel scalability of the program. Calotoiu, et.al. [15] construct a performance model structured by the call tree of the program and based on data gathered by Scalasca [16]. This performance model supports approximate optimization at the function level.

There is, however, no previous paper which defines, implements and evaluates, a set of workflows which automates solving the specific problems of optimizing at the task and loop levels and supporting the transition from task level parallelism to hybrid (task + threaded) parallelism.

There are many tools that can execute some steps of one or more of the workflows presented in this paper. In with regards to performance measurements, tools such as HPCToolkit [4], PAPI [17], Intel® Vtune™ Amplifier [18], TAU [19] and Open|SpeedShop [20] provide hardware performance counter based measurements on the CPU, Intel Phi coprocessor, and NVIDIA GPUs. For parallel applications, tools like Score-P [21] and Extrae [22] provide a powerful instrumentation environment to measure the performance of MPI, OpenMP, and CUDA codes. IBM and Allinea offer commercially available tools that will provide some or most of the information.

ACKNOWLEDGMENT

The authors gratefully acknowledge National Science Foundation support under award number ACI-1134872 (Stampede).

REFERENCES

[1] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "Perfexpert: An easy-to-use performance diagnosis tool for hpc applications," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.41

[2] A. Rane and J. Browne, "Enhancing performance optimization of multicore/multichip nodes with data structure metrics," *ACM Trans. Parallel Comput.*, vol. 1, no. 1, pp. 3:1–3:20, May 2014. [Online]. Available: http://doi.acm.org/10.1145/2588788

[3] L. Fialho and J. Browne, *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems, pp. 261–277. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07518-1_17

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput.: Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[5] S. Plimpton, C. Battaile, M. Ch, L. Holm, A. Thompson, V. Tikare, G. Wagner, X. Zhou, C. G. Cardona, and A. Slepoy, "Crossing the mesoscale no-mans land via parallel kinetic monte carlo," 2009.

[6] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[7] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995.

[8] C. Dave and R. Eigenmann, "Automatically tuning parallel and parallelized programs," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 126–139. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13374-9_9

[9] A. Aiken and A. Nicolau, "Optimal loop parallelization," *SIGPLAN Not.*, vol. 23, no. 7, pp. 308–317, Jun. 1988.

[10] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua, "Restructuring fortran programs for cedar," *Concurrency: Practice and Experience*, vol. 5, no. 7, pp. 553–573, 1993.

[11] S. W. Kim, I. Park, and R. Eigenmann, *Languages and Compilers for Parallel Computing: 13th International Workshop, LCPC 2000 Yorktown Heights, NY, USA, August 10–12, 2000 Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ch. A Performance Advisor Tool for Shared-Memory Parallel Programming, pp. 274–288.

[12] I. Park, N. H. Kapadia, R. J. O. Figueiredo, R. Eigenmann, and J. A. B. Fortes, "Towards an integrated, web-executable parallel programming tool environment," in *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, J. Donnelley, Ed. IEEE Computer Society, 2000, p. 9.

[13] Z. Pan, B. Armstrong, H. Bae, and R. Eigenmann, "On the interaction of tiling and automatic parallelization," in *OpenMP Shared Memory Parallel Programming - International Workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12-15, 2006. Proceedings*, ser. Lecture Notes in Computer Science, M. S. Müller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, Eds., vol. 4315. Springer, 2005, pp. 24–35.

[14] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 47:1–47:12.

[15] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, November 2013, pp. 1–12.

[16] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[17] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. Washington, DC, USA: IEEE Computer Society, 2000.

[18] J. Reinders, *VTune Performance Analyzer Essentials*, 1st ed. Hillsboro: Intel Press, 2005.

[19] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006.

[20] M. Schulz, J. Galarowicz, D. Maghrak, and W. Hachfeld, “Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis,” *Scientific Programming*, vol. 16, no. 2–3, pp. 105–121, 2008.

[21] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir, pp. 79–91.

[22] “Extrae website,” https://www.bsc.es/computer-sciences/extrae, [Online; accessed March-2016].