Ayguadé, E. [et al.]. Is the schedule clause really necessary in OpenMP?. "Lecture notes in computer science", Juny 2003, vol. 2716, p. 147-159. The final authenticated version is available online at https://doi.org/10.1007/3-540-45009-2_12

Is the *schedule* clause really necessary in OpenMP?

Eduard Ayguadé[1], Bob Blainey[2], Alejandro Duran[1], Jesús Labarta[1], Francisco Martínez[1], Xavier Martorell[1],Raúl Silvera[2]

[1] CEPBA-IBM Research Institute
Departament d'Arquitectura de Computadors
Universitat Politcnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain.
{*eduard, aduran, jesus, fmartin, xavim*}*@ac.upc.es*
[2] IBM Toronto Lab
8200 Warden Ave
Markham, ON, L6G 1C7, Canada
{*blainey,rauls*}*@ca.ibm.com*

Abstract. Choosing the appropriate assignment of loop iterations to threads is one of the most important decisions that need to be taken when parallelizing Loops, the main source of parallelism in numerical applications. This is not an easy task, even for expert programmers, and it can pontentially take a large amount of time. OpenMP offers the *schedule* clause, with a set of predefined iteration scheduling strategies, to spedify how (and when) this assignment of iterations to threads is done. In some cases, the best schedule depends on architectural characteristics of the target architecture, data input, ... making the code less portable. Even worse, the best schedule can change along execution time depending on dynamic changes in the behaviour of the loop or changes in the resources available in the system. Also, for certain types of imbalanced loops, the schedulers already proposed in the literature are not able to extract the maximum parallelism because they do not appropriately trade–off load balancing and data locality. This paper proposes a new scheduling strategy, that derives at run time the best scheduling policy for each parallel loop in the program, based on information gathered at runtime by the library itself.

1 Introduction

Parallel loops are the most important source of parallelism in numerical applications. OpenMP, the standard shared–memory programming model, allows the exploitation of loop–level parallelism thorough the `DO` work–sharing and `PARALLEL DO` constructs. Iterations are the work units that are distributed among threads as indicated in the `SCHEDULE` clause: `STATIC`, `DYNAMIC` and `GUIDED` (all of them with or without the specification of a `chunk` size). While in a `STATIC` schedule the assignment of iterations to threads is defined before the computation in the loop starts, both `DYNAMIC` and `GUIDED` do the assignment dynamically

as the work is being executed. In `DYNAMIC` threads get uniform chunks while in `GUIDED` chunks are progressively reduced in size in order to reduce scheduling overheads at the begining of the loop and favour load balancing at the end.

Deciding the appropriate scheduling of iterations to threads may not be an easy task for the programmer, specially when it depends on dynamic issues, such as input data, or when memory behaviour is highly dependent on the schedulie applied. Load unbalancing or high cache miss ratios, respectively, are usually symptoms of inappropriate iteration assignments. In OpenMP, the programmer can play with the predefined schedules mentioned above or embed its own scheduling strategy in the application code if none of them is appropriate. The `chunk` size (or number of contiguous iterations assigned to a thread) is a parameter that needs to be appropriately set in order to avoid non–friendly memory assignments of iterations and/or excessive run-time overheads in the process of getting work. Even worse, the decisions may depend on parameters of the target architecture (going against performance portability, one of the key issues in OpenMP).

The standard offers the possibility of specifying that the loop needs to be serialized if a certain condition is met (`IF` clause in OpenMP). Some OpenMP runtime systems can also decide to serialize the execution if certain conditions (e.g. loop bounds, number of threads, ...) are met.

In order to decide a schedule strategy, some simple rules of thumb are usually applied: `STATIC` for those loops with good balance among iterations; unbalanced loops should use an interleaved schedule (`STATIC` with `chunk`) or some sort of dynamic schedule (`DYNAMIC` or `GUIDED`). However, the use of dynamic schedules usually incurs high scheduling overheads and its non-predictive behaviour tends to degrade data locality (non–reuse of data across loops or multiples instances of the same loop). Although these rules work for a large number of simple cases, they are far from complete and can lead to poor decisions. Other schedules need to be built by the user, embedding code and data structures to implement them.

In this paper we will present a proposal to remove such burden from the programmer by letting the runtime decide which is the most appropiate schedule for a given loop. In the next section we motivate the work by using a simple unbalanced and applying different schedules. In section 3 we present the generic framework of our work. In section 4 we describe our current prototype implementation. In section 5 we show the results obtained with some benchmarks. Finally in section 6 concludes the paper and shows future directions of research.

2 Motivation and related work

In order to motivate the proposal presented in this paper, we will consider a synthetic loop in which the cost of each iteration is $cost(i) = k/i$, k being a parameter that depends on the number of iterations of the loop. This distributes almost all the weight of the loop to the first iterations (the first 1% of the iteration space accounts for 50% of the cost of the loop). During the execution, each thread accesses a matrix indexed with its thread identifier. Therefore, the loop only has

temporal locality. The loop has been executed 500 times on a 4-way IBM Power4 system. Figure 1 shows the results obtained for different schedules.

In this loop, using a `STATIC` schedule leads to a highly unbalanced execution, with a speedup of 1.64 with respect to the execution with one thread. Although the use of a `STATIC` schedule with a chunk size of one increases the speedup to 1.97, it still does not achieve good balance. For example, if $k = 10000$, the work of the first thread is 1.6 times the work of the fourth. Therefore, this is an example in which it seems appropriate to use either a `DYNAMIC` or `GUIDED` schedule. Using a `DYNAMIC` schedule we get a speedup of 1.82 due to the high scheduling overheads and degradation of temporal locality. A `GUIDED` schedule, which usually tends to reduce these scheduling overheads, is decreasing the speedup to 1.63; this is due to the fact that some threads get excessively large chunks at the begining that are not well balanced with the remaining ones.

Probably, the best schedule would be ad–hoc trying to reduce scheduling overheads, optimize load balancing, avoid false sharing or a combination of these issues that compensate them. However, in other cases it may be even impossible for the the programmer to calculate this "ideal schedule" because it depends on variables only available at runtime (architecture, input data, interaction between loops or processes, ...).

Other schedules, similar in nature to `GUIDED`, have been proposed in the literature, such as trapezoid scheduling [1], factoring [2] and tapering [3]. These schedules are variations of the previous ones and are tailored for certain load unbalance patterns. Some other schedules try to take in account the geometric form of the iteration space. For example folding is a variation of `STATIC` in which iterations i and $N - i$ are assigned to the same thread, N being the total number of iterations. For the previous synthetic example, this schedule is unable to improve the behaviour achieving a speedup of 1.77. A study of the most suited schedule for different loops, grouped by their iteration execution time variance is presented in [4].

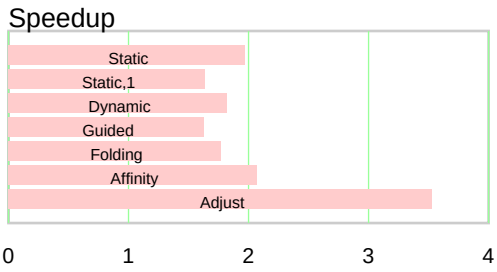


Fig. 1. Speedup for different schedules on a 4-way IBM Power4

Other proposals try to achieve load balancing by applying work stealing. They usually assign iterations to threads in a `STATIC`–like manner; in Affinity Scheduling (AS) [5] threads steal chunks of work from other processors as soon as they finish with the initially assigned work. This work stealing adds a dynamic part to the work assignment that does not favour memory behaviour. Affinity Scheduling is available in the IBM OpenMP runtime system as a non–standard feature that can be specified in the `OMP_RUNTIME` environment variable. In our synthetic example, affinity scheduling achieves the highest speedup of all the available schedules (2.07). Dynamically partitioned affinity scheduling (DPAS) [6] learns from work stealing in order to derive a new `STATIC`-like schedule, to be used in subsequent instances of the loop, in which each thread has a different chunk size. Other proposals such as Feedback Guided Dynamic Schedule (FGDS) [7] and Feedback Guided Load Balancing [8] avoid the dynamic part by simply measuring the amount of unbalance (without applying work stealing) and derives a similar `STATIC` schedule. In [9] a different approach is used where a processor is reserved to compute partial schedules based on the load of each processor that are placed on the processors work queues.

The main objective of this paper is to advance one step further in the use of dynamically derived schedules and show how they can optimize the behaviour of real applications. We propose a general framework oriented towards having a self–tuned OpenMP runtime system and show an implementation on a real commercial system. The runtime is able to characterize the execution of a loop and learn from past executions in the same run in order to gradually enhance the assignment of iterations to threads. The objective is to relieve the user from the task of deciding the best schedule for each loop and ideally lead to better performance. For instance, in the same synthetic example described above, our proposed framework achieves an speedup of 3.53. The scheduling is achieved in a completely transparent way with no additional specification from the programmer in the source code.

3 Dynamic derivation of loop schedules

In order to decide the most suited schedule, the runtime needs to collect information that characterizes the behavior of the loop. Although the compiler could provide static information derived from the analysis of the source code (or even could be provided by the user as hints), such as the initial schedule for the loop or the identifiers of other loops with similar memory access and/or workload patterns, most of the information can only be gathered at runtime. Our main goal is to show that this information gathering, loop characterization and optimization can be done at runtime with minimal (or no) information from the user and/or compiler and with reasonable (or even negligible) overhead.

At runtime, the information that can be dynamically observed and collected includes: size and bounds of the iteration space, variation in the cost of the iterations, memory access patterns and conflicts in the access to memory containers (cache lines or pages), etc. In the process of observing these metrics, granularity

is an important issue to consider: Overall per–thread execution time versus execution time for iteration (or groups of iterations), overall per–thread cache miss ratio (or page fault ratio) versus detailed correspondance between cache misses (page faults) and loop iterations, ... The accuracy level (granularity) may not be constant during the execution of the program and vary according to the characterization process itself. The first time a loop is executed, or after detecting a high perturbation in its current characterization, the runtime could switch to fine–grain measurement status. Once the runtime detects a stable characterization, it could switch to a coarser–grain mode of operation, in order to minimize unnecessary overheads.

When no information is available for a loop (e.g it is the first time the loop is executed), the runtime could start with a predefined schedule (or even the one suggested by the programmer) and try to characterize the loop doing fine-grain measurements. Another possibility could be to adopt the characterization for a another loop (for which a characterization has been done) and make fine-grain measurements. This characterization reuse may be important in order to reduce the time required to reach a stable characterization state. Reuse hints could also be provided by the programmer or the compiler (e.g. providing information about affine loops). It could even be possible that the runtime discovers affinity relationships between loops (i.e. loops whose characterization is the same or changes in the same way) that are executed inside an iterative sequential loop.

Our belief is that the use of work-stealing strategies during the characterization process should be avoided in order to prevect perturbations with the characterization process itself. For example, work stealing adds a dynamic part to the assignment of iterations that may worsen memory locality and increase memory latencies both for the stealing and stolen threads. However, in some cases this extra overhead may be compensated with load balancing, thus reducing the overall time to reach a (new) efficient schedule for the loop. Loops that are executed only once could also have a better behaviour if work stealing is applied.

Based on the available characterization, and in order to decide the best suited assignment of iterations to threads, the runtime should try to:

- Preserve spatial locality, by assigning contiguous chunks of iterations to the same thread whenever possible. This will optimize the access to memory containers (cache lines or pages) and reduce the likelihood of false sharing.
- Preserve temporal locality, by reusing the same schedule in subsequent execution instances of the same loop (or an affine one). This will favour data reuse.
- Balance loops, so that all threads get the same amount of work; this does not imply the same amount of iterations.

Once the scheduling is decided, the characterization process continues in order to detect further opportunities for refinement. As mentioned before, and since this information gathering could have a significant impact on performance, the runtime should be able to switch to the most appropriate granularity level, depending on the status of the characterization itself.

Up to this point, we have not addressed possible interferences between the schedules applied to different loops. The use of different iteration assignments in different loops may degrade memory locality and be counter–productive. To this end, the characterization process could consider sequences of loops and derive decisions that optimize the behaviour of the sequences and not the individual loops.

4 Current Implementation

In this section, we describe the current prototype for the self–tuning OpenMP runtime system that has been implemented in the XL IBM Runtime. In the description we consider both the characterization and the decision processes. In this prototype implementation, mainly load balancing issues are addressed.

The runtime identifies each parallel loop instance with a tuple $\{L,\ IS\}$. The first component (L) of this tuple identifies the loop in the program (using the pointer to the routine generated by the compiler that encapsulates the loop). The second component (IS) identifies different instances of the same loop (using the iteration space of the loop: iteration limits). Whenever possible, the information derived by the runtime for a tuple $\{L_i,\ IS_j\}$ will be re-used to initially characterize other tuples $\{L_i,\ IS_k\}$ that correspond to the same loop. All tuples that correspond to the same loop L_i summarize the past behaviour for that loop.

For each tuple $\{L,\ IS\}$ the following information is recorded:

- The pointer to the routine that identifies the loop.
- The iteration space description.
- The $\{L,IS\}$ balancing information.
- The last subchunk information gathered by the runtime for the tuple (See below).
- The relation of weights between iterations. In the current implementation only two patterns are handled: constant weight, when all iteration have the same weight, and unkown. Others patterns could be recognized if their properties are useful for scheduling.
- The last schedule applied to the tuple.

The balance information is composed of:

- A state that indicates the actual knowledge of the balance of the {loop/iteration space}.The possible states and their meaning are sumarized in table 1.
- The number of consecutive executions that this balance state has been maintained.
- The actual definition of balanceness for the tuple, i.e the percent of unbalance allowed. This definition varies among time. When there is no knowledge about balance the limit is 10% of imbalance. Later on, as there is more confidence the limit is increased first to 20% and later to 25%. The increment of our definition of balance enables to elude minimal perturbations of the system.

State	Meaning
Unknown	No balance information is known yet.
Unbalanced	Runtime found that it is unable to balance the tuple.
Balanced	Runtime found a schedule that balanced the tuple.
Highly balanced	Runtime feels really confident that the schedule applied will be balanced.

Table 1. Possible balance states

The transitions of the balance state are shown in figure 2. If no information is inherited from other states, the first balance information is the *Unkown* state. When a balanced execution is done a transition to the *Balanced* state is done. After N unsuccessful executions the *Unbalanced* state is achieved. While in this state a balanced execution, normally due to a change in behaviour, takes a transition to the *Balanced* state. While in the *Balanced* state any imbalanced execution reverts to the *Unkown* state. N balanced executions in the *Balance* state increases the confidence on the decision and goes to the *Highly Balanced* state. An unbalanced execution in this last state reverts to the *Balanced* state. Note that when there is confidence in the balance of the loop there have to be two consecutive unbalanced executions in the last N to revert from *Highly Balanced* to *Unkown*. This gives some tolerance to perturbations while being able to adapt to changes in the behaviour. The actual value for N is 10 times.

When a {loop/iteration space} is executed for the first time a new state is allocated for it. If it is also the first execution of the loop the state is initialized to an *Unkown* balance state and a hipothesis that the iterations weights are constant is tried. If other iteration spaces were executed for the same loop before, the initilization is inherited from the most similar iteration space. In other words, the balance information, the iteration information and a modified version of the schedule applied to the other iteration space (adding or substracting iterations) are copied.

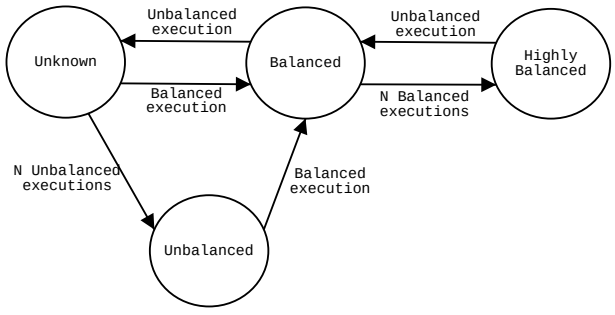


Fig. 2. Balance information transitions

Every time the loop starts the execution in a given iteration space, the schedule to be used (and its parameters) are decided. This decision is based on the current state of the tuple. Currently, one of two the following schedules can be choosen:

- *OpenMP* `STATIC`.
- *Non–uniform* `STATIC`. This schedule is very similar to the previous one. Each thread is also assigned a chunk of contiguous iterations that are determined prior to the loop execution. However, chunks assigned to threads may be of different size. When the size of each chunk is properly chosen a very good load balance is achieved. Temporal and spatial localities are achieved as in the `STATIC` case because of the assignment of contiguous iterations and schedule reuse.

The scheduling decision is sumarized in table 2. When something is known about the balance of the loop, either that loop is balanced or that it is unbalanceable, the last schedule applied is reused. In case the loop is considered balanced this schedule will be the the one that achieved the balance. In case the loop is considered unbalanceable the schedule will be the best schedule found which will be used there on. When nothing is known about the balance of the loop, either because a proper schedule that balances it has not been found yet or because it hasn't reached the threshold to give up, the schedule used is static if the iterations were found to be constant, otherwise an non-uniform static schedule is used with the assignment of iterations for each thread calculated based on previous gathered measurements.

Balance state	Iteration cost	Schedule
Unkown	Constant	Static
Unkown	Non-constant	Non-uniform Static
Other	*	Reuse previous

Table 2. Schedule decision function

The assigment of iterations for each thread when the non-uniform static schedule is used works as follows. First, the weight each thread should have is calculed dividing the total time by the number of threads. Afterwards, subchunks are assigned sequentally to the first thread. When the sum of the subchunks is greater that the estimated weight per thread, the last subchunk is broken assuming all iterations in the subchunk have the same weight, and the number iterations of the first thread is adjusted in consequence. The remaining iterations of the last subchunk are left to be assigned to other threads. Next, we start assigning iterations to the second thread, and so on. If we arrive to the last thread there are still some iterations left are assigned to the last thread.

Also when going to execute the loop, the granularity of measuring has to be decided. Two granularities are supported: subchunk (fine granularity) and

thread (coarse granularity). When subchunk granularity is used, to avoid excesive overhead of measuring every single iteration, groups of iterations called subchunks are measured. The number of these subchunks is variable for each thread and depends of the number of iterations that have been assigned. When thread granularity is choosen, the measures are done for the overall iterations of each thread (so there is only one subchunk per thread). The measures right now include only execution times. The decision of choosing between the two granularities is taken based on the balance state of the loop as shown in table 3.

Balance state	Granularity used
Unkown	Fine measuring
Other	Coarse measuring

Table 3. Time measures granularity decision function

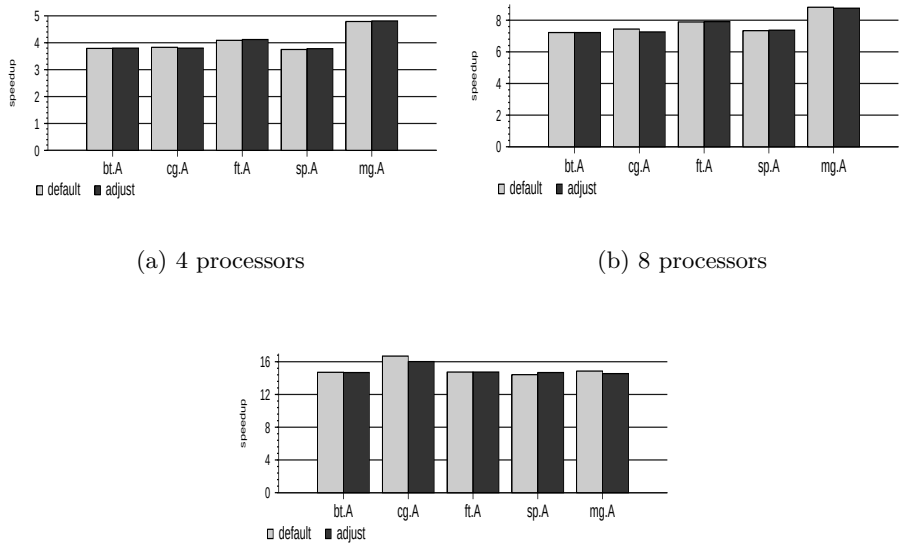
After the execution of the loop a new state has to be generated. It is calculated using the actual state and the measures taken from the execution. If the execution time of each thread does not deviate from the average more than the actual definition of balance the execution is considered balanced, otherwise the execution was unbalanced. With this information a transition in the balance state automaton is done. Also, based on the taken measures, if the mean iteration weight per thread does not deviate from a certain threshold the iterations are considered to be constant, otherwise iteration weights are calculed from subchunk information. Finally, current schedule decision are saved as the last schedule applied. If this schedule also resulted in the best schedule applied so far it is saved as the best schedule used.

With this runtime environment, loops that would require the use of `STATIC`, `DYNAMIC`, `GUIDED` or even other schedules not available in OpenMP (such as folding) can be efficiently executed, as shown in the evaluation section.

5 Evaluation

In order to evaluate the proposed schedule, we have used some programs from the SPEComp suite [10] (swim, ammp, gafort, apsi, wupwise and art), class A NAS OpenMP benchmarks [11] (bt, ft, cg, sp and mg), and a computational kernel that calculates the legendre polynomial. They are OpenMP versions that make use of the `SCHEDULE` clause.

When a *parallel do* loop does not specify an schedule (using the `SCHEDULE` clause) it defaults to a special value: `RUNTIME`. This value means that the actual schedule to be used can be specified in the environment variable In order to specify a schedule not specified in the standard, we use `OMP_RUNTIME`. If this variable is not specified the schedule used is implementation dependent, thought tipically is `STATIC`. Two different kind of tests have been run: the first run



(a) 4 processors

(b) 8 processors

(c) 16 processors

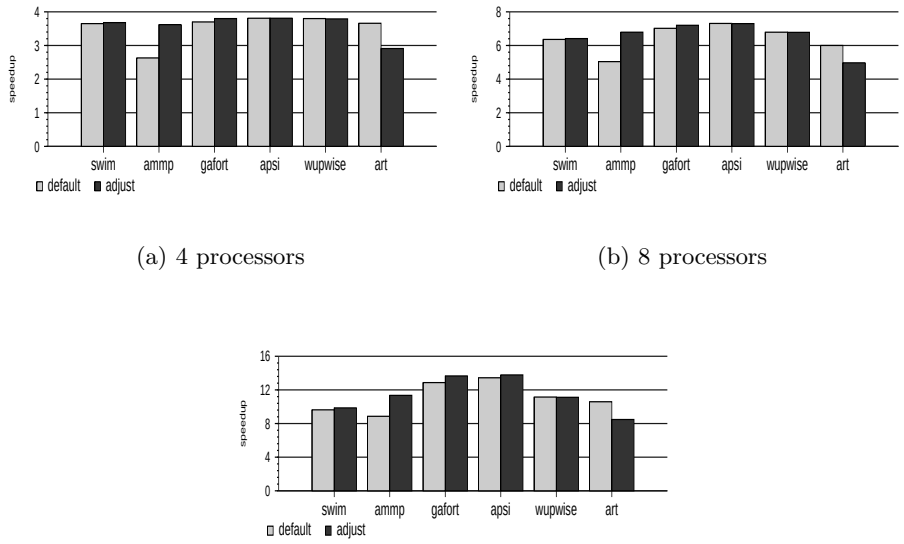
Fig. 3. Speedups for the NAS benchmarks

with the schedules originally set in the benchmark, and `OMP_RUNTIME` defined as `STATIC`. In the second test, we eliminated the `SCHEDULE` clauses of all parallel loops and set the `OMP_RUNTIME` environment variable to `ADJUST`, specifying that the schedule described in Section 4 should be applied.

The benchmarks were run in a p690 32-way Power4 [12] machine at 1.1 Ghz with 128 Gb of RAM. We used the IBM's XLF compiler with the `-O3 -qipa=noobject -qsmp=omp` flags, and the operating system was AIX 5.1 .

Programs *bt*, *ft*, *cg*, *sp*, *mg*, *swim*, *apsi* and *wupwise* use `STATIC` schedules. The programs *ammp*, *gafort* and *art* use `GUIDED` schedules. The *legendre* kernel has two triangular loops that are programmed with a "folding" schedule embedded in the application code. In *mg*, is interesting to note that the iteration space changes from one execution to another.

Figure 3 shows the results for the NAS benchmarks, on the x-axis are the benchmarks and on the y-axis is the speedup achieved for the default schedule and for the adjust runtime. It can be seen that the results obtained from both methods are almoust equivalent (the difference is always below 5%). This is due the NAS benchmarks have loops very balanced with good locality by default and there is little room for improvement here. The important thing is that our mechanism is able to decide also a `STATIC` schedule for this type of loops, that are quite common, with a negligible overhead in reaching that decision.



(a) 4 processors

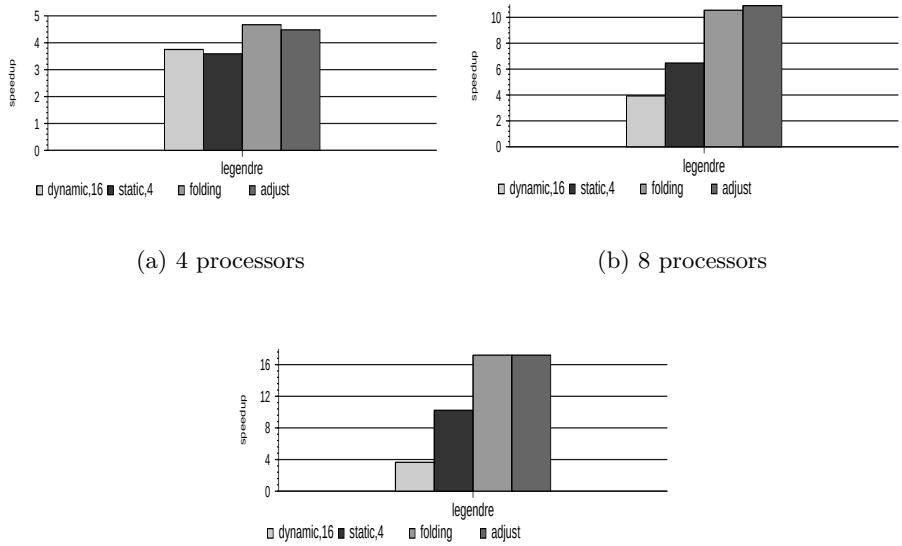
(b) 8 processors

(c) 16 processors

Fig. 4. Speedups for the SPEComp benchmarks

Figure 4 shows the results for the SPEComp benchmarks. Notice that the ones that use `STATIC` schedules achieve the same performance with diferences below 3%. Of the programs that use `GUIDED` schedules, *ammp* is really improved by our method (27% with 4 processors, 22% with 16 processors). The reason for this is that the *non–uniform static* schedule derived by the runtime has much better locality, because iterations are executed contiguously and the schedule is reused, than the original schedule. The improvement of *gafort* is negligible (below 4%). As *gafort* makes random vector acceses thus we do not obtain the locality gains seen in *ammp*. The *art* benchmark is the worse case our method can find, as the main code is a loop executed just one time, and we cannot make use of the knowledge obtained in that first execution. As we see, our method needs loops that are executed iteratively. To solve this cases a schedule that is more flexible to imbalanced codes than `STATIC` should be used the first time (such as the Affinity schedule), but right now the original code performs a 20% better that our proposal.

In figure 5 are the results for the *legendre* kernel results. In addition to the default folding schedule and adjust, the best OpenMP schedules found (dynamic,16 for 4 processors and static,4 for 8 and 16 processors) are also shown. Note that one user that decided to use as schedule *dynamic,16* after some basic analysis in a minor configuration would be fooled if later he would run it in



(a) 4 processors

(b) 8 processors

(c) 16 processors

Fig. 5. Speedups for the Legendre kernel

a high end production system. In any case, our proposal is able to find a *non–uniform static* schedule that performs as well as the "hardwired" folding schedule (differences are below 5%) and much better than any OpenMP schedule (from 16% to 41% better).

6 Conclusions and Future Work

The paper investigates the feasibility of having a self–tuned OpenMP runtime system.

A general framework and description of the characteristics of such system have been presented. A prototype implementation of the runtima has been also described, and although there still much work to do, the results presented are encouraging as cover a good range of typical loops and show that it is possible to dynamically derive schedules based on information gathered and characterization done at runtime.

Under such environment, the `SCHEDULE` clauses specified by the user should be seen as hints that could help the runtime system to arrive to a stable characterization in a fast way.

Future work will follow two directions. The first one is to increse the number of loop characteristics recognized by the runtime by analyzing many applications.

This way, characterization of a broad range of loops would be posible. Work will cover the following topics:

- Exploration of metrics other than execution time.
- Detection of affinity relations between loops based on the loops execution history. With that information scheduling of groups of loops will be posible.
- Detection of patterns in the iteration spaces. This will reduce the memory usage of the runtime by grouping information of different iteration spaces and to provide better scheduling for the loop by issuing a schedule compatible with all of the iteration spaces.

The second line of work is directed to improve the implementation of the runtime to achieve a faster characterization mechanism by eliminating the penalties caused by the poor memory behavior during the process of finding the appropriate schedule. Also giving the posibility of issuing schedules that do not use all available threads, if this leads to a better performance, will be explored.

Also, preliminary work has been started to explore the benefits of the self–tuned runtime in applications that run on imbalanced resources, for example, under the Power4 processor having an application running with 3 threads (2 threads on the same chip and 1 in the other).

Acknowledgements

Authors want to thank Julita Corbalan for her insightful comments. This work has been supported by the IBM CAS program, the POP European Future Emerging Technologies project under contract IST-2001-33071 and by the Spanish Ministry of Science and Education under contract TIC2001-0995-C02-01.

References

1. T.H. Tzen and L.M. Ni. Trapezoid self-scheduling scheme for parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, 1993.
2. E. Schonberg S.F. Hummel and L.E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.
3. S. Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of ACM SIGPLAN'92 Conference on Programming Language Desing and Implementation*, pages 220–211, 1992.
4. Kelvin K. Yule and David J. Lilja. Categorizing parallel loops based on iteration execution time variances. Technical Report HPPC-94-13, University of Minnesota, 1994.
5. E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. Technical Report TR410, 1992.
6. S. Subramaniam and D.L. Eager. Affinity scheduling of unbalanced workloads. In *SuperComputer'94 Conference Proceedings*, 1994.
7. J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *European Conference on Parallel Processing*, pages 377–382, 1998.

8. Francis H. Dang and Lawrence Rauchwerger. Speculative parallelization of partially parallel loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.
9. Babak Hamidzadeh and David J. Lilja. Self-adjusting scheduling: An on-line optimization technique for locality management and load balancing. In *International Conference on Parallel Processing*, pages 39–46, 1994.
10. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. *Lecture Notes in Computer Science*, 2104, 2001.
11. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
12. Steve Behling et al. The POWER4 processor introduction and tuning guide. Technical Report SG24-7041-00, International Technical Support Organization, November 2001. ISBN 0738423556.