John von Neumann Institute for Computing

Scheduling issues on IBM p690: Performance Analysis with the PARbench Environment

H. Dietze, W.E. Nagel, B. Trenkler

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 717-724, 2006.

http://www.fz-juelich.de/nic-series/volume33

Scheduling issues on IBM p690: Performance Analysis with the PARbench Environment

Heiko Dietze[a], Wolfgang E. Nagel[a], Bernd Trenkler[a]

[a]Center for Information Services & High Performance Computing (ZIH) Dresden University of Technology D-01062 Dresden, Germany

1. Introduction

This paper investigates scheduling properties of parallel programs on IBM p690. Based on the PARbench environment, it is the continuation in a series of research studies to analyze different multiprogramming scheduling issues on high performance computer systems ([1], [3], [4], [5]). First, the *PARbench* environment is introduced briefly. Then the performance analysis main results, as the basis for the scheduling study, are described. Finally, the behavior of the scheduling system is discussed.

2. PARbench

The PARbench benchmark system is designed to simulate virtually every workload the user might specify. It can execute many benchmark programs in parallel and record their behavior with regards to time flow. This feature is applied analysing a system workload containing parallel programs in a multiprogramming mode.

The basis for each workload program generated with PARbench is a set of synthetic kernels. All kernels follow two rules: First, they are short enough to combine into one workload. Second, together they are able to represent the overall parameter space of the used computer system.

The used version of PARbench provides 17 different kernels, each consisting of three parts: writing to disk, a loop nest, and reading from disk. The I/O-Part can be varied in 5 stages, including the option to skip it. The loops are often derived from scientific calculations but do not always solve a problem. Every kernel can be used with different data sizes by changing the size of the used matrices. In result, there are 340 different kernel versions available. As a concept for parallelization, OpenMP is used. The number of threads per job during execution is a parameter specified by the user.

3. IBM p690

The tested IBM p690 is a shared memory system with 32 processors. Based on the IBM POWER4+ processor with a clock rate of 1.7 GHz, it has a theoretical peak performance of 6.8 GFLOPS per processor.

3.1. Preliminary Investigations

The first step for the work with the PARbench benchmark is to identify the kernel properties. This includes, amongst other things, the FLOP-rate and rate of the processors memory references per second (MREFS). The results represent the available parameter space for the scheduling experiments, particularly for the CPU and memory utilization.

The review of the results for all 340 Kernels shows that very few kernels reach a rate of more than 25% theoretical peak performance (1700 MFLOPS) and yet fewer reach more than 50% theoretical

peak performance (3400 MFLOPS). The same, but to a lesser degree, can be said about the rate of memory references. A plot of all kernels is available in Figure 1. Importantly, a high FLOP-rate cannot be achieved in conjunction with a high rate of memory references. This behavior is due mainly to the processor and memory interface, but the PARbench benchmark properties can be of influence.

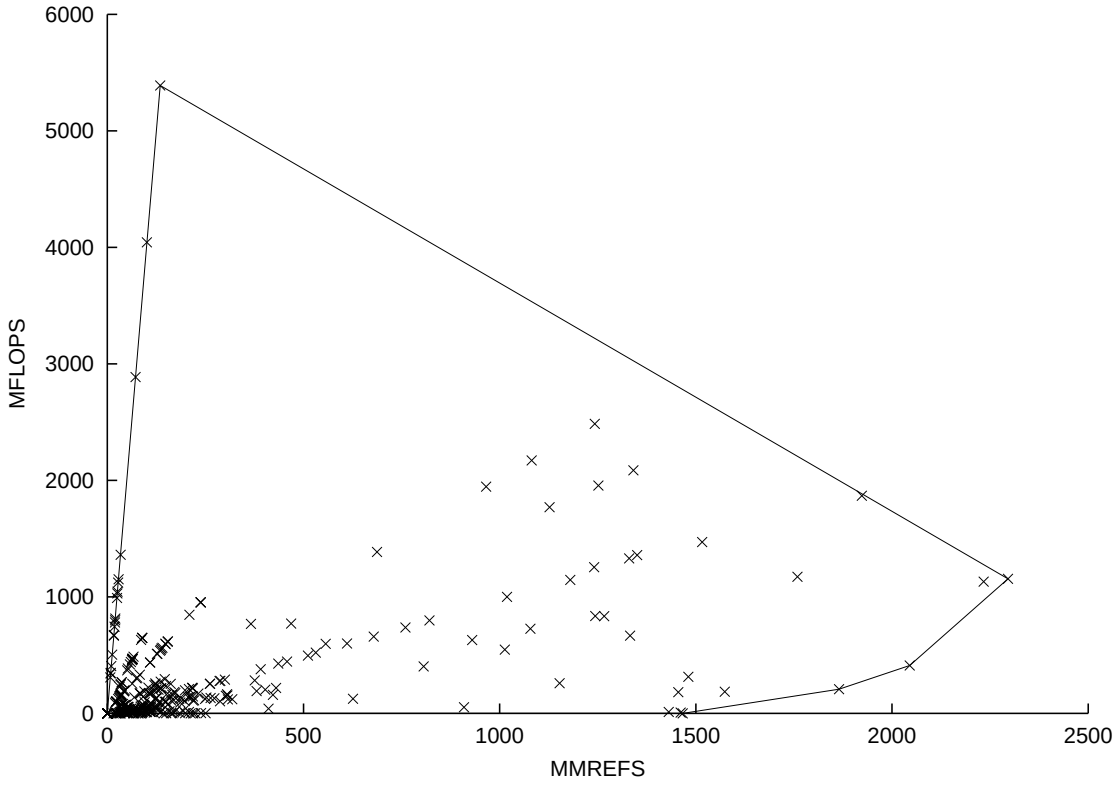

Figure 1. PARbench kernel performance (MMREFS and MFLOPS) on IBM p690

To reach the theoretical floating-point peak performance, it is necessary to use both floating-point-units continuously with FMA-Operations (Fused Multiply Add Operation). This is not possible in all PARbench-Kernels. This disables the attainable rate down to 1.7 GFLOPS. Even if it was possible, insufficient bandwidth and high latencies to the memory and level-three (L3) cache prevent a better performance. The design of the processor core also reduces the attainable FLOP-rate. Managing out-of-order execution via groups is a design decisions. Tracking by groups of instructions and not individual instructions reduces the necessary hardware, but introduces additional overhead such as the cost of creating an instruction group and the cost due to lost opportunities for parallel computation.

Another problematic design feature is the two floating-point-units. Each unit has a six staged pipeline. However, each pipeline stage needs an independent floating-point instruction to operate continuously. Thus, 12 or more instructions are needed to feed the pipeline in an continuous and efficient way. This can be a problematic requirement for loops with data dependencies (even in one iteration). The peak performance of 5.4 GFLOPS (Kernel number 1) can only be achieved by using 40 FMA-register-register-operations per iteration of the inner loop.

Another aspect is the processors memory interface. There are two load/store-units per processor core. Every unit can issue one load or store operation per clock cycle. Thus the theoretical limit for the memory references is 3800 million (3800 MMREFS). Consequently, each floating-point-unit is only provided with one operand from memory (or caches) per cycle.

The results indicate that the memory subsystem is a bottleneck of the system. The high latency of the level-three (L3) cache and structural problem of the shared level-two (L2) cache increase this problem. Thus, only 2.3 GMREFS (Peak 3.8 GMREFS) could be achieved with PARbench kernel number 186. This kernel uses the level-one (L1) data-cache very efficiently, because it uses 15 out of 16 entries of a cache line and the hardware pre-fetching can be employed.

4. Scheduling on the IBM p690

Scheduling is the task of the operating system. The tested IBM p690 system is equipped with the AIX 5L Version 5.2 for POWER. It uses a thread based scheduling system with priority queues (256 stages). The scheduling algorithm is a fair round robin algorithm with dynamic priorities. The priorities are calculated on the threads CPU usage. Since AIX 4.3.3, each processor has its own queue (32 queues for 32 processors). Furthermore, there is one global queue for all processors available. However, this queue must be explicitly activated and overwrites the system of local queues.

4.1. Serial Workloads

The initial experiments investigate the basic scheduling behavior. This includes workloads based on kernel number 1 and 186. The jobs based on kernel 1 represent the ideal workload without side effects. A run with 32 identical jobs based on kernel 1 and with 32 processors is nearly perfect. All 32 jobs run without any waiting time. There is virtually no additional user-time and the system-time, an indicator for the schedulings overhead, is negligible.

The same experiment, with the 32 identical jobs, based on the memory intensive kernel 186 results differently. There is a visible prolongation of the user-time for several jobs. This is explained by the interaction of the jobs when caches are shared. When caches are shared, more time is needed during loading and storing.

Both experiments show that scheduling serial workloads in full-load situations are unproblematic. However, cache interference and memory must be watched, as it can increase the necessary user-time.

4.2. Serial Workloads with Overload

The experiments in overload situations are simulated by using 48 jobs based on kernel 1. Each job has a initial runtime of 100 seconds. The operating system offers two variants for organizing the scheduling: one queue per processor, which is the standard version (results shown in Figure 2), or one global queue for all processors (not shown). The scheduling algorithm for all queues is the fair round robin system with dynamic priorities.

The results from the system of 32 queues show two classes of jobs: with and without waiting time. Naturally, slight waiting time due to the overload is to be expected. The reason for the difference is the late migration between the processor queues. If a processor is idle, one of the waiting jobs migrates to the other processor. In the beginning of the experiment with 48 jobs there are queues with only one job and queues with two jobs. The single jobs run without interruptions until completion. Meanwhile, two jobs share one processor fairly via round robin. Only after completion of one quasi exclusive jobs is there a migration (late migration). Now the two jobs can complete without interruption.

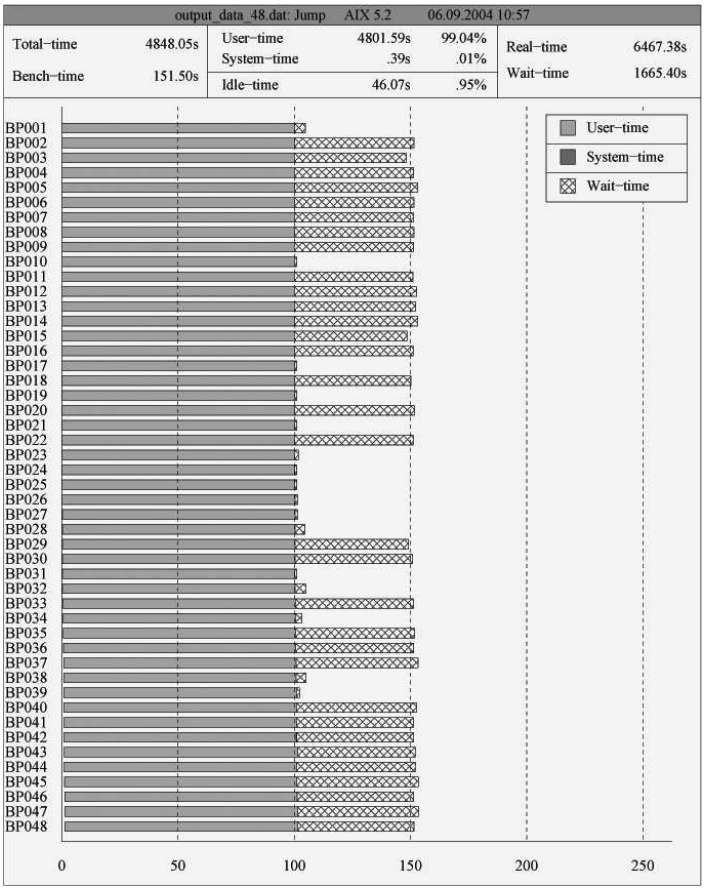

Figure 2. PARbench experiment with 48 serial jobs based on kernel 1 running on 32 processors with one queue per processor.

Using the global queue for scheduling, the result is quite different. All jobs are distributed equally among the 32 processors by the round robin scheduling algorithm. This means every job has now a waiting time of approximately half the runtime. Due to the loss of several processors exclusive use, the overall waiting time increases. The global queue in conjunction with a fair scheduling concludes in an increase of waiting time.

4.3. Mixed Workloads

For experiments with parallel workloads, we create a new set of 32 jobs. Every job is created with the goal of 1900 MFLOPS, 1000 MMREFS and 100 seconds runtime. This is impossible with one special kernel, so a sequence of kernels are used. This sequence contains different kernels to approximate the given job properties.

As an example of a mixed workload with both parallelized and serial jobs, we use 16 serial jobs and 16 parallel jobs with 4 threads each. The jobs are divided into four groups, each with 4 serial and 4 parallel jobs. The result for the system of one queue per processor is shown in Figure 3 and for the global queue in Figure 4.

The system of local queues produce only a small speedup, but this is done at a high cost. The overall user-time increases and the waiting time is not reduced. Therefore, there is no actual gain in runtime. This is due to the migration between the 32 queues of the 32 processors.

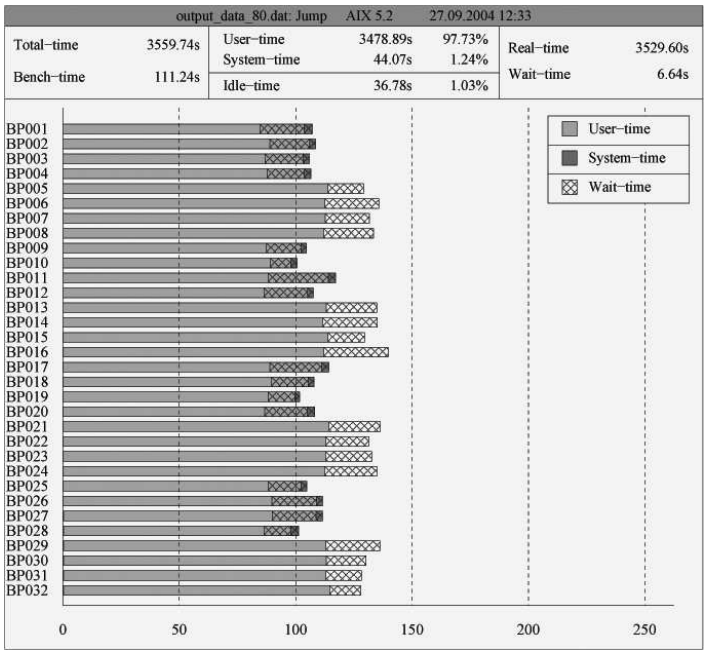

Figure 3. PARbench experiment with 16 jobs a 4 threads and 16 serial jobs running on 32 processors with one queue per processor.

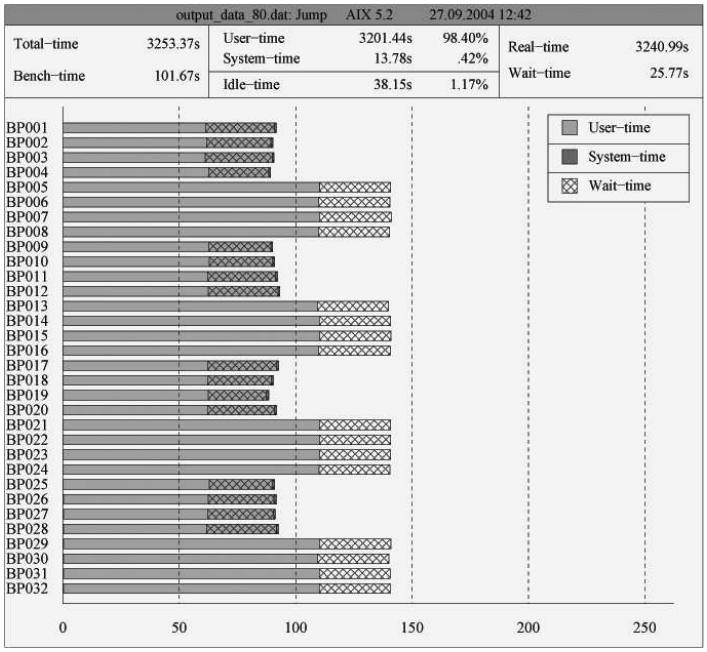

Figure 4. PARbench experiment with 16 jobs a 4 threads and 16 serial jobs running on 32 processors with the global queue.

With initiation, every job is assigned to a queue and processor. Then, the 16 jobs start to execute their parallel regions (OpenMP). Still, the 4 threads of one parallel job remain mainly on the same processor. The late migration reduces the ability to work with threads of the same job in parallel. A consequence of the increased busy-wait is additional user-time.

The results from the global queue (Figure 4) show a better acceleration than the local queues. Also, there is no additional overall user-time. However, due to the properties of the global queue, there is an increase of the overall waiting time. This annuls any gain due to parallelization in comparison to the serial working of the set of jobs. The two possibilities for organizing scheduling do not produce convincing results in an overload situation with mixed workloads of parallel and serial jobs.

During the experiments with mixed workloads, we encountered some shortcomings in the accounting system, particularly with the user-time. In result, the accounting system recorded less user-time for parallel programs (OpenMP) and thus more to the sequential programs. In extreme cases, we measured sequential programs with a higher user-time than real-time. Yet, the sum of all user-times remains correct. This seems to occur only if thread-parallel and sequential programs shared the same queue. Thus, the drawn user-time in Figures 3 and 4 for parallel jobs is, in reality, higher and for sequential jobs, smaller. With parallel-only or sequential workloads, there are no such visible effects.

The experiment with a parallel-only workload uses a set of 32 parallel jobs with 4 threads per job. The results with local queues are shown in Figure 5. There are no gains visible and the user-time

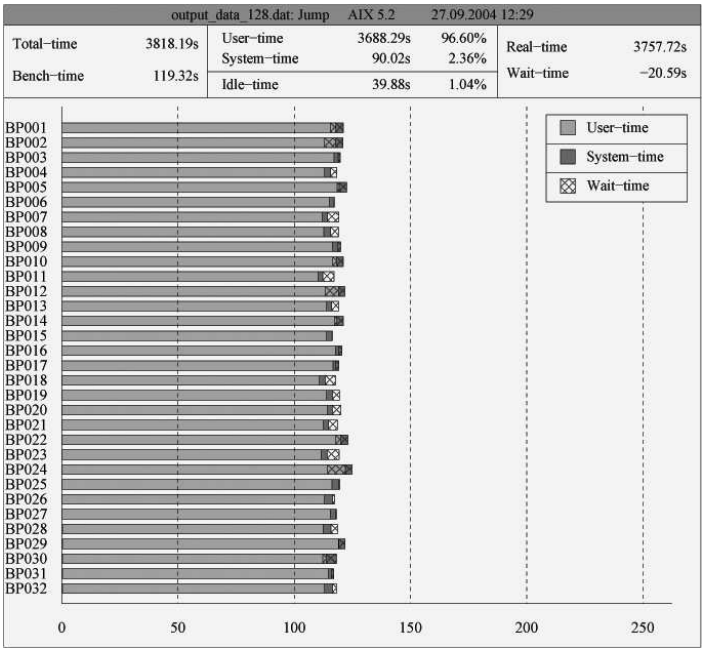

Figure 5. PARbench experiment with 32 jobs a 4 threads with one queue per processor.

increases even further. The effect of one queue per processor is most visible here: no thread has a chance to work in parallel with a thread of its own job because no thread can migrate to another processor due to late migration. The increase of user-time is explained due to the busy-wait during parallelization.

Changing to the global queue (not shown) could stop the additional user-time, but there is still no gain through parallelization. Serializing the threads is nearly ideal and this case is viewed as almost identical to the calculation with 32 serial jobs. The only cost is the additional work for the scheduler, which can be measured in an increase of the system-time.

4.4. Handoptimized Workload

The main goal of parallel work is to gain an acceleration due to reduced runtime. The current results show that a system of local queues have a negative influence on parallel computation. The global queue is slightly better but it also increases the overall waiting time thus negating any positive effects.

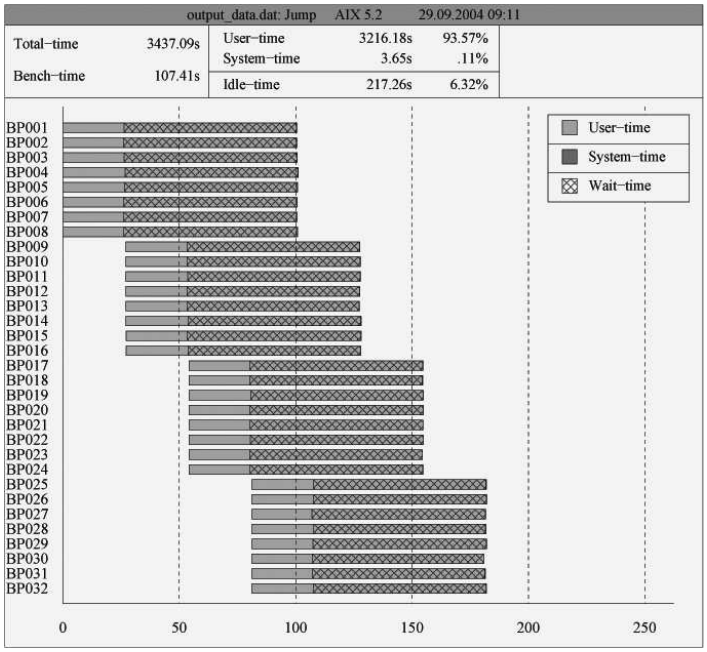

Figure 6. PARbench experiment with 32 jobs a 4 threads and hand-optimization delay in the beginning of parts of the workload.

To avoid the overloads negative effects, one can use a delay when starting some of the jobs in the workload. This is done, for instance, manually, as shown with PARbench in Figure 6 (for the example of 32 jobs with 4 threads per job).

This results in nearly ideal behavior. Every job can gain a considerable acceleration and nearly all jobs finish earlier than in a sequential mode. Only the last group of 4 jobs completes after the 100 second limit. The reason being the acceleration is slightly less than ideal for a parallelization with 4 processors. But this is negligible in contrast to the results without a manual optimization (ref. Figure 5).

A manual optimization or the use of a workload manager to prevent an overload situation contradicts the ideal multiprogramming mode. Additionally, the use of MPI in a cluster of IBM p690 nodes is only efficient if all processes of one MPI-job are active at the same time. This can not be guaranteed in an overload situation with multiprogramming.

5. Summary and Outlook

The performance results show that the POWER4+ Processor has several deficits reaching its peak performance. The main reasons are the design of the processor core and the memory-interface. The caches can not compensate this problem. The level-three (L3) cache with its high latency and the shared level-two (L2) cache are also problematic factors.

The operating systems scheduling system works perfect for workloads up to a load of 100 percent. In overload situations, such as multiprogramming environments, both scheduling variants exhibit drawbacks. Using the global queue can produce acceptable results but at the cost of higher waiting times. The method of one queue per processor produces the best results with serial workloads. Its disadvantage being the longer process-time when used in conjunction with parallel jobs. Additionally, the speedup is small or the parallel runtime is longer than the sequential version. A manual optimization using a start-delay to decrease the workload could produce the best results. Furthermore, we have detected some shortcomings in the accounting system which, just by accident, record fewer to the parallel programs.

In the future, limited degree of parallelism will be tested. It is expected, that multiprogramming provides better results in this situation, as it is the mostly used dedicated mode. Furthermore, other machines e.g. Sun Fire E25K, NEC-SX8 will be tested.

6. Acknowledgment

We would like to thank the John-von-Neuman Institute at the Research Center Juelich for providing the access to their IBM p690 Cluster JUMP thus making the measurements possible.

References

[1] Sebastian Boesler. Performance-Analyse von Hochleistungsrechnern im Multiprogramming-Betrieb: Untersuchungen auf der SGI Origin. Diploma-Thesis, Dresden University, Dec 2001.

[2] Heiko Dietze. Das PARbench-System: Untersuchungen zum Scheduling von parallelen Programmen auf der IBM p690. Diploma-Thesis, Dresden University, Nov 2004.

[3] Klaus Fabian. Leistungsuntersuchungen von Multiprozessorsystemen auf der Basis des parametergesteuerten Lastbeschreibungssystems PAR-Bench unter besonderer Bercksichtigung von parallel ablaufbaren Teillasten. Technical Report Jl-2671, Forschungszentrum Jlich, August 1992.

[4] Andreas Kowarz. Performance-Untersuchungen mit dem PARbench-System auf unterschiedlichen Parallelrechnern. Diplomarbeit, Technische Universitt Dresden, Fakultt Informatik, Institut fr Technische Informatik, Mai 2003.

[5] Wolfgang E. Nagel and Markus A. Linn. Benchmarking parallel programs in a muliprogramming enviroment: The PARbench system, 1991.