Characterizing Performance of Applications on Blue Gene/Q

Paul F BAUMEISTER [b] Hans BOETTIGER [a] Thorsten HATER [b,1]
Michael KNOBLOCH [b] Thilo MAURER [a] Andrea NOBILE [b] Dirk PLEITER [b]
Nicolas VANDENBERGEN [b]

[a] *IBM Deutschland Research & Development GmbH, Germany*
[b] *JSC, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany*

Abstract. Recently the latest generation of Blue Gene machines became available. In this paper we introduce general metrics to characterize the performance of applications and apply it to a diverse set of applications running on Blue Gene/Q. The applications range from regular, floating-point bound to irregular event-simulator like types. We argue that the proposed metrics are suitable to characterize the performance for a larger set of computational science applications running on today's massively-parallel systems. They therefore do not only allow to assess usability of the Blue Gene/Q architecture for the considered (types of) applications. They also provide more general information on application requirements and valuable input for evaluating the usability of various architectural features, i.e. information, which is needed for future co-design efforts aiming for exascale performance.

Keywords. Performance characterization, parallel algorithms, Blue Gene

Introduction

With the introduction of the Blue Gene/Q architecture the available concurrency has been significantly increased at various levels compared to the predecessor architecture. Most notably, a multi-core processor with up to 64 threads of execution has been introduced. Additionally, the width of the SIMD unit has been doubled to four double precision numbers. To understand how well relevant scientific applications can be mapped to this new architecture we developed an extensive set of performance characterization metrics.

The process of implementing applications can be modeled as a two-step procedure. First, a formal solution of the problem is chosen and mapped onto a set of computational tasks. Second, for each computational task a particular implementation is derived, which is then compiled into computer code for a specific architecture. For the analysis, we will typically focus on those tasks, which form a performance critical region, i.e. kernels. We distinguish between metrics primarily

[1] Corresponding author: `t.hater@fz-juelich.de`

related to computation, communication and synchronization. In the framework of this paper we will however only present results for the computation metrics.

Additionally, task metrics and implementation metrics are differentiated. The former category includes any metric, which only depends on the definition of the computational tasks and possibly a (simple) machine model. It has to be derived from a theoretical analysis of the task, i.e. cannot be measured. A typical example is the mandatory information flow between processor and external memory. On the other hand, an implementation metric is defined depending on both hardware architecture as well as on system software (e.g. compilers) and possibly the input data. Such a metric is typically obtained by direct measurement, for example via performance counters. An example for a computation-implementation metric is the fraction of instructions dealing with floating-point arithmetic.

An implementation metric is either deterministic or stochastic. The former refers to metrics that are reproducible for given input data and executable, like the number of load-store instructions. For stochastic metrics the measured values may change for each run due to the non-deterministic behaviour of the hardware. A typical example is the number of loaded cache lines.

For all considered applications quasi-realistic problems were identified in cooperation with the developers. Major contributions to runtime due to communication are briefly discussed where important. In general, we tuned the number of processes per node and number threads per process to a local optimum.

1. Blue Gene/Q Architecture

The BG/Q compute chip [1] combines 16 cores running at 1.6 GHz in a single package together with a network unit and memory controllers. The aggregated peak performance is 204.6 GFlop/s. The bandwidth to main memory has a maximum of 42.7 GB/s of which 80 % are achievable for an even mix of reads and writes. An additional core is provided for OS purposes, but inaccessible to the programmer. This system-on-a-chip design allows for high energy efficiency and very low OS jitter. Each core is equipped with an L1 cache of 16 kB and all cores share a 32 MB L2 cache. The L2 cache is versioned to support transactional memory operations and speculative execution of threads. Each core comprises a prefetch unit, which supports different types of prefetching, e.g. stream prefetching.

Each of the compute cores supports four-way simultaneous multi-threading (SMT). There are two instruction pipelines per core, one in the execution unit *XU* and one in the auxiliary execution unit *AXU*. The latter is connected to a separate register file and offers a 4-way SIMD floating-point pipeline with a throughput of four double-precision multiply-add operations per clock cycle. In each clock cycle two instructions from different threads can be issued to any of the pipelines, i.e. at least two threads are required to keep both pipelines filled.

2. Applications

We chose a set of three applications developed at Forschungszentrum Jülich, based on their prior investment in the Blue Gene/P platform and scalability. Additionally, diversity of application performance signatures was a major motivation.

KKRnano: An application from solid state physics based on density functional theory (DFT) [2]. The application is implemented in Fortran and hybrid-parallelized using MPI and OpenMP. It features a 'traditional HPC program' structure in the sense that it operates on large, regular arrays of floating point numbers. MPI parallelization is based on a regular domain decomposition.

The central operation is the iterative solution of the sparse linear system arising from the KKR approach, which is implemented by the quasi-minimal residual (QMR) method [9,10,11]. We expect this application to be bound by floating point performance and thus benefit heavily from vectorization and multi-threading. This is supported by the high arithmetic intensity found in the later analysis.

PEPC: Second, we analyse PEPC, a mesh-free parallel tree-code for computation of long-range Coulomb or gravitational forces in N-body particle systems [3] in the framework of the Barnes-Hut algorithm [12]. This application is used for simulations in different fields, in particular plasma physics. In this Fortran based code both MPI and Pthreads are used for parallelization.

Despite depending on floating-point performance, PEPC differs from KKR-nano in so far as it uses trees of particles instead of large regular arrays. The irregular memory access poses challenges for multi-threading and vectorization.

NEST: The final application is a graph-based discrete event simulation for neural networks [4]. Compared to the first two candidates it is radically different in many ways. It is written in C++ and parallelized with MPI and OpenMP (or Pthreads). The core of NEST is an interpreter for a custom language for representing neuronal networks. Depending on the problem set or program, the performance characteristics may differ strongly. Due to the nature of the problem, we expect minimal reliance on floating-point capabilities and a large fraction of integer operations and branches.

Although NEST is available as Open Source Software, we are using an internal preview version, tuned for large scale HPC systems.

3. Metrics

We present a compact subset of the metrics we use to characterize the performance of computational science applications. As noted earlier we restrict ourself to computation metrics for this paper. For a more extensive set of metrics see [5]. While we applied our metrics to the Blue Gene/Q architecture, most can easily be applied to other architectures as well.

The *task metrics* are shown in the upper part of Table 1. Here we assume a very simple node model, namely a processing unit attached to an arbitrarily sized main memory. We furthermore assume the processing unit to offer a sufficiently

Metric (Task)	**Description**
IF-mem-cpu	Information flow from (off-chip) between memory and processing unit
NO-FP	Number of floating-point operations
AI	Arithmetic intensity = NO-FP / IF-mem-cpu
Metric (Implementation)	**Description**
IM-fpa	Fraction of floating-point arithmetic instructions
IM-vfpa	Fraction of vector floating-point arithmetic instructions
IM-ia	Fraction of fixed-point arithmetic instructions
IM-ldst	Fraction of load-store instructions
IM-br	Fraction of branching instructions
ρ_{fp}	Useful floating-point operations vs. operations performed in hardware
$\rho_{\mathrm{ld,st}}$	Number of loaded vs. stored Bytes
$\rho_{\mathrm{int,fp}}$	Number of loaded plus stored Bytes for integers vs. floats
t_{WC}	Wall-clock time [cycles]
$C_{\mathrm{x,XU}}$, $C_{\mathrm{x,AXU}}$	Instruction throughput per execution pipeline [instr/cycle]
C_{fp}	Floating-point operation throughput [Flop/cycle]
C_{mem}	Memory interface throughput [Byte/cycle]
ρ_{L1d}	Number of hits in L1 data cache vs. cacheable data load operations
ρ_{pre}	Number of hits in prefetch buffer vs. cacheable data load operations
ρ_{L2}	Number of hits in L2 data cache vs. cacheable data load operations

Table 1. Task and implementation-computation metrics.

large cache such that only mandatory cache misses occur, i.e. data re-use is maximized. The *implementation metrics* are listed in the lower part of Table 1. All metrics can be measured on Blue Gene/Q using a simple (proprietary) software interface to the performance counters. The data is aggregated over a single node (unless stated otherwise).

As of today, most computational science applications that need HPC compute resources require the execution of a large number of floating-point operations. The arithmetic intensity (AI) puts the number of floating-point operations (NO-FP) in relation to the minimum amount of data, which needs to be transferred between processor and external memory (IF-mem-cpu). The latter is natural as memory access became the dominant performance factor for many codes. Depending on whether the arithmetic intensity is large or small, the achieved floating-point operation throughput C_{fp} or the memory bandwidth C_{mem} are primary performance indicators. It is, however, important to look at the numerator (NO-FP) and denominator (IF-mem-cpu) separately to assess how close the actual implementation is to this idealized case. These can be compared to the product of throughput and wall-clock time t_{WC}.

To understand the instruction throughput it can be enlightening to consider the instruction mix. It allows to assess how well the different execution pipelines can be filled. On Blue Gene/Q all floating-point instructions are executed by the AXU pipeline, while all others are issued to the XU pipeline. Depending on whether the number of instructions to be scheduled to any of the pipelines is significantly larger, the overall performance depends on the corresponding instruc-

tion throughput metric $C_{\mathrm{x},i}$ (i=XU, AXU; $C_{\mathrm{x},i} \leq 1$). $C_{\mathrm{x},i} \approx 1$ for all i indicates the processor's micro-architecture being well balanced for a given application.

To put the actual number of instructions, e.g. floating-point instructions, in relation to the number of operations, e.g. the task metric number of floating-point operations (NO-FP), we have to analyse how well the application can be mapped to the given instruction set architecture (ISA). A floating-point instruction dispatched to the AXU pipeline can result in up to 8 floating-point operations. The parameter ρ_{fp} measures the number of useful versus the maximum number of floating-point operations that could have been performed using the same number of instructions. ρ_{fp} may be small because the SIMD unit is not exploited by the application, e.g. due to data alignment restrictions. In this case the fraction of vector floating-point instructions (IM-vfpa) would be significantly smaller than the fraction of floating-point instructions (IM-fpa).

To characterize the memory access patterns generated by the application we consider the ratio of the number of Bytes loaded to stored, $\rho_{\mathrm{ld,st}}$, as well as the ratio of loads and stores of fixed-point to floating-point numbers, $\rho_{\mathrm{int,fp}}$. A ratio of $\rho_{\mathrm{ld,st}} > 1$ indicates that the application maps well on a processor architecture where read and write performance are different. In case of Blue Gene/Q processor the maximum aggregate bandwidth provided by the crossbar switch connecting the L2 cache slices and the cores is asymmetric: The aggregate maximum read and write bandwidth from and to the L2 cache is 409.6 and 153.6 GByte/s, respectively [1]. For floating-point intensive applications $\rho_{\mathrm{int,fp}}$ might be expected to be small. However, indirect memory access due to complex data structures may result in a large number of integer load operations, thus leading to large $\rho_{\mathrm{int,fp}}$ and additional pressure on the memory subsystem.

To analyze the use of the memory hierarchy we do not only consider the usual cache miss rates, ρ_{L1d} and ρ_{L2}, but also the efficiency of the stream pre-fetcher. For this purpose we monitor the number of hits in the pre-fetcher buffer normalized by the total number of cacheable loads ρ_{pre}.

4. KKRnano

KKRnano is the application most accessible to static compiler optimization. We present the analysis of a test-case consisting of 512 atoms of a Nickel-Palladium alloy, which can be considered as a small but still typical use case. We used 32 nodes at 16 processes per node and four OpenMP threads per process. At these node counts KKRnano exhibits good strong scaling properties for the test-case at hand. Weak scaling extends to much larger node counts.

Kernels We identify two kernels, each contributing around 10% of the total runtime for the given test-case. We work at a level of logically distinct tasks, i.e. functional units with a single call-path.

The first kernel, `sprszmm`, performs a sparse matrix-vector product $\Lambda_{ij} \cdot \gamma_j$, where Λ_{ij}, $\gamma_i \in M_{n\times n}(\mathbb{C})$ and in each of the M rows N elements of Λ are non-zero. This operation is the central part to the iterative QMR solver.

The second kernel is the core of a block circulant pre-conditioner called by the QMR solver. Since both kernels are exhibiting the same central characteristics, we focus on the former.

Metric	KKRnano (16/4) **sprszmm**	PEPC (4/16) **treewalk**	NEST (4/16) **update**
IF-mem-cpu [Bytes]	$2.5 \cdot 10^{11}$	$> 2.0 \cdot 10^{12}$	–
NO-FP	$1.0 \cdot 10^{12}$	$> 3.1 \cdot 10^{12}$	0
AI	4	1.6	0
IM-fpa	0	0.04	0.08
IM-vfpa	0.34	0.02	0
IM-ia	0.05	0.14	0.13
IM-ldst	0.37	0.31	0.46
IM-br	0.06	0.05	0.12
ρ_{fp}	0.98	0.38	0.14
$\rho_{\mathrm{ld,st}}$	2.41	1.92	2.12
$\rho_{\mathrm{int,fp}}$	0.23	0.98	3.26
t_{WC}	$2.99 \cdot 10^{10}$	$2.9 \cdot 10^{8}$	$1.95 \cdot 10^{11}$
$C_{\mathrm{x,XU}}$	0.51	0.19	0.63
$C_{\mathrm{x,AXU}}$	0.25	0.03	0.03
C_{fp}	31.7	5.5	1.6
C_{mem}	9.5	0.13	3.4
ρ_{L1d}	0.79	0.94	0.94
ρ_{pre}	0.15	0	0
ρ_{L2}	0.05	0.06	0.06

Table 2. Task and implementation metrics results using 32 Blue Gene/Q nodes. The optimal process to thread ratio for the used test-case is given as $\left(\frac{N_{\mathrm{MPI}}}{N_{\mathrm{node}}} / \frac{N_{\mathrm{thr}}}{N_{\mathrm{MPI}}}\right)$.

Task metrics Working from the definition of `sprszmm` we compute the task metrics. Independent of the concrete implementation, the minimum number of Bytes we have to move from main memory to processor is the aggregated size of the operands $\mathbf{\Lambda}$ and $\boldsymbol{\gamma}$: $16NMn^2+16Mn^2$ Bytes. To store the result, we have to write $16Mn^2$ Bytes back to memory. Actually computing this result requires $8MNn^3$ operations on floating-point numbers in the limit of large matrices. The parameters N, M and n depend on the number of atoms, the considered interaction range and the cut-off of the potential expansion. In the implementation considered here the task has been changed in order to make use of the zgemm library routine. This results in additional $16MNn^2$ Bytes to be loaded and stored and therefore significantly lower arithmetic intensity.

Implementation metrics Table 2 gives an overview of the measured runtime metrics. The kernel is dominated by complex, double-precision linear algebra, as seen in the relation of IM-ia to IM-(v)fpa. The floating-point instructions are almost exclusively in the vectorized form, as the compiler is able to unroll most loops. This is due to the fact that in the current version of the code every iteration count is known at compile-time. The number of measured floating-point operations is very close to the task metric NO-FP that was computed in the limit of large matrices. Branches and integer operations account for about 10% of the instructions. This overhead is caused by address computations and loop management. The rate of instructions issued per cycle per core is around 0.75. Due to the amount of

memory accesses the load on the XU pipeline is much higher than for the AXU pipeline.

The prefetcher and first level cache cover around 93% of the memory accesses. Due to the sparseness of the matrices and their memory layout it is to be expected that stream prefetching is inefficient. This is consistent with the observed low hit rate ρ_{pre}. The total amount of data transferred across the memory interface is only about 10% higher than expected.

Communication Collective reduction and gather operations involving all processes are the dominating communication patterns. For the test-case they account for about 20 % of the execution time, split between Allgather 73%, Allreduce 16% and Broadcast 11%. This number looks larger than it is, since KKRnano is not homogeneously parallelised and these timings include waiting times for threads, which are not part of the current computation. Note that this is inherent to the problem and not a flaw of the implementation.

5. PEPC

As indicated above, the irregular memory access in PEPC is a major issue of the optimization. Most time is spent in the graph traversal organizing the branches and leafs of the octree according to the multipole acceptance criteria. As a test-case we use an input set with 250,000 particles randomly distributed with a homogeneous density as inital positions running three time steps on 32 nodes with 4 MPI processes each. Per process 16 threads had been started of which one manages all MPI communication while the remaining threads share the computational task as worker threads.

Kernels The kernel of PEPC considered here is named `treewalk` and takes about 88 % of the total runtime with the given test input decks. It incorporates the floating-point intensive subroutine `forcecalc` with 27 % and performs integer operations, indexing and pointer chasing during the other 61 %. The task metrics have been evaluated for `forcecalc` only, while the implementations metrics refer to `treewalk` including `forcecalc`.

Task metrics The subkernel `forcecalc` performs about 244 (useful) double precision floating-point operations and loads about 160 Bytes each time it is called. The graph tasks hardly perform any floating-point operations. The number of (integer) loads in this part of the application is difficult to predict since several structures such as the translation of keys to addresses involve inverse searches, pointer chasing and list lookups.

Implementation metrics PEPC achieves a relatively low instruction throughput as $C_{\mathrm{x,XU}}$ and $C_{\mathrm{x,AXU}}$ are small. This indicates a large number of stall cycles, despite the low branch count, and as we can see from the IM-vfpa metric, effectively no vectorization. Most instructions are integer loads and stores we can see from IM-ldst and $\rho_{\mathrm{int,fp}}$. The number of cacheable loads exhibits a hit rate in L1 of 94 %. Performance counters show the ratio of L2 hits over misses to be close to 300. Also the high ratio of IM-ldst over C_{mem} shows that the tree walk is largely memory latency bound.

Communication As the node count increases, an increasing amount of runtime is due to collective communication, while point-to-point communication decreases, intersecting near 32 to 64 nodes. The largest amount of time is spent within the buffered send and the matching receives, however, the total time spent in MPI routines is small compared to the kernel execution times for these (small) system sizes. Only the communicator threads performs a busy wait in `MPI_Iprobe` polling for the worker threads to finish.

6. NEST

The core of NEST's algorithm is the distributed update of neuron states. Every MPI process collects the outgoing neural spikes over a finite time interval Δt and communicates them to the respective addressees using MPI. Each incoming spike is then used to alter the respondent's state. The concrete nature of the alteration depends on type and state of the respective neuron. Event transport and generation are modeled as a stochastic process.

The nature of the problem is basically a look-up and update in a distributed graph. Depending on the neuron type(s) the update may involve floating-point computations, but not in significant amounts. Since we need to look-up neuron types and state as well as the synapses – the graphs edges – the load is balanced different from for example KKRnano.

The performance counter data was gathered using 32 nodes with 4 MPI processes per node and 16 OpenMP threads per process. As performance of NEST is typically maximized by filling as much memory as possible, additional threads are favored over MPI processes in production runs as it helps to reduce the memory footprint. We decided to emulate this behaviour, although the problem size at hand does not require it.

Kernels The core part of NEST is the `update` method, which dispatches incoming events to their local receivers. Dispatch is handled dynamically depending on the types of the transporting synapse, the event and the receiving neuron. In order to complete the delivery, NEST has to determine these types, which is done by leveraging virtual method dispatch of the C++ runtime. Additionally, the containers aggregating the respective instances have to be traversed. After the event has been routed to its recipient, the state of the corresponding neuron is updated.

From this, we can expect the dominant characteristics to be integer operations for indexing and address calculations and conditional branches, both from container iteration and virtual function dispatch.

Task metrics The task we are considering involves almost no floating-point operations, so the corresponding estimates – including AI – are approximately zero. Due to the stochastic nature of the simulation, as timings and numbers of events are random, we can give also no *a priori* estimate of the data movement.

In general, this makes it almost impossible to derive reliable task metrics. However, we roughly estimated the instruction mix to be distributed as 20% branches, 30% fixed point arithmetic and 50% load/store operations from the source code.

Implementation metrics From the definition of the `update` kernel we expect strong reliance on the memory capabilities of the architecture and as most operations involve indirection and indexing, mainly the memory (load) latency. This is reflected in the high amount of memory operations in the instruction mix. Indirection and indexing show mainly in the high amount of integer operations. Our prediction for the instruction mix on the source code level exhibits a reasonable fit with the measured values.

We find good coverage by the first level data cache of around 94%. However the prefetching mechanisms are unable to handle the irregular traversals of the data structures, there are simply no streams to exploit. There are also no repeating access patterns, which would allow to use the list prefetcher.

The XU pipeline executes by far the most of the instructions, i.e. the application would likely benefit from multiple scalar pipelines. Obviously, the architecture provides more floating-point operation performance than is needed for this application, as at most 10% of the instructions are floating-point related. More complicated neuron models may, however, lead to a somewhat higher floating-point operations intensity.

7. Related work

In this paper we consider a set of metrics, which reflect on the one hand the application dependent performance signatures and on the other hand the hardware characteristics of the architecture on which the application is executed. The focus therefore differs from a set of papers where a performance characterization is suggested largely independent of the architecture, e.g., in order to classify parallel applications. We follow, however, one such attempt of application classification, [6], to classify the proposed metrics. Also the Apex project [7] aims for a performance characterization independent of particular hardware architectures. There the assumption is made that performance behaviour can be characterized by a small set of architecture independent performance factors. We did not restrict us to architecture independent performance metrics in order to assess the usability of architectural features, like SIMD units or memory prefetch engines. We consider it however instructive to relate architecture independent task metrics and implementations metrics to assess whether the architecture is balanced for a given application and to quantify limiting factors. The authors of the roofline model [8], e.g., proposed to determine an operational density to derive upper bounds for the performance for a given memory bandwidth.

8. Summary and Conclusions

In this paper we have presented a subset of metrics we defined in order to characterize the performance of applications on a given architecture. Here we focussed on Blue Gene/Q, the latest generation of massively-parallel HPC architectures from IBM, which scales to dozens of petaflops. We applied our performance characterization methodology to a set of three increasingly irregular scientific applica-

tions. All of them are in the need for petascale computing performance and had been optimized for this architecture.

For most of the applications kernels considered floating-point operations are not dominating. As a result the high performance of the SIMD unit could not be exploited for these applications. These could possibly benefit from additional scalar processing pipelines as the instruction throughput for the scalar pipeline ($C_{\mathrm{x,XU}}$) was found to be much higher than for the SIMD pipeline ($C_{\mathrm{x,AXU}}$). The instruction mix was typically dominated by load and store operations.

We observed efficient use of the upper levels of the memory hierarchy, which is partially due to the relatively large L2 cache, that allows to reduce the pressure on the interface to the external memory. Only one of the applications considered here, KKRnano, could exploit the memory prefetching capabilities.

By applying (a selected subset of) our performance metrics to different applications running on Blue Gene/Q we obtain both information on the application's performance signatures as well as on the hardware characteristics of the considered architecture and on the usability of architectural features.

Acknowledgements

This work has been performed in the framework of the "Exascale Innovation Centre" which is partially supported by the state of Nordrhein-Westfalen. We would like to thank other members of the EIC and the developers of the applications discussed in this paper for their support, in particular: J. Eppler, W. Homberg, L. Arnold, R. Zeller.

† IBM and Blue Gene are trademarks of IBM in USA and/or other countries.

References

[1] IBM Blue Gene team, "Design of the IBM Blue Gene/Q compute chip," IBM Journal of Research and Development 57 (2013), 1:1-1:13.
[2] R. Zeller, "Linear scaling for metallic systems by the Korringa-Kohn-Rostoker multiple-Scattering method," Challenges and Advances in Comp. Chemistry and Physics 13, 2011.
[3] M. Winkel et al., "A massively parallel, multi-disciplinary BarnesHut tree code for extreme-scale N-body simulations," Comp. Phys. Comm. 183 (2012) 880.
[4] M.-O. Gewaltig, M. Diesmann, "NEST (NEural Simulation Tool)," Scholarpedia Vol. 2, No. 4 (2007).
[5] P. F. Baumeister et al., "Analysis of scientific applications on Blue Gene/Q," TR (2013).
[6] A. S. van Amesfoort et al. Sips, "Metrics to characterize parallel applications," 2010.
[7] E. Strohmaier, H. Shan, "Architecture independent performance characterization and benchmarking for scientific applications," in: Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (2004).
[8] S. Williams, A. Watermann, D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," Comm. ACM, Vol. 52, No. 4 (2009).
[9] W. Kohn and N. Rostoker, "Solution of the Schrödinger equation in periodic lattices with an application to metallic lithium," Physical Review, Vol. 94, No. 5 (1954)
[10] J. Korringa, "On the calculation of the energy of a Bloch wave in a metal" Physica, Vol. 13, No. 6 (1947)
[11] R. Freund and N. Nachtigal, "QMR: a quasi-minimal residual method for' non-Hermitian linear systems," Numerische Mathematik, Vol. 60, No. 1 (1991)
[12] J. Barnes and P. Hut, "A hierarchical O(NlogN) force-calculation algorithm," Nature 1991