ALGORITMA *PARALLEL SUPERVISED PNN STRUCTURE DETERMINATION* DAN IMPLEMENTASI BERBASIS *MESSAGE PASSING INTERFACE*

Heru Suhartanto dan Herry

Fakultas Ilmu Komputer, Universitas Indonesia, Depok, Indonesia
heru@cs.ui.ac.id

Abstrak

Probabilistic Neural Network (PNN) adalah salah satu tipe jaringan neural yang umum digunakan untuk memecahkan permasalahan klasifikasi pola. Di samping struktur jaringan dan metode pelatihan yang sederhana, PNN memiliki kelemahan utama yaitu dalam menentukan struktur jaringan yang terdiri dari penentuan nilai parameter *smoothing* dan jumlah neuron yang digunakan pada lapisan pola. Dengan adanya kelemahan ini, beberapa peneliti mengajukan algoritma *Supervised PNN Structure Determination* (SPNN) dengan tujuan untuk mempermudah penentuan struktur PNN. Akan tetapi dalam implementasi iteratif yang telah dilaporkan, SPNN masih memerlukan waktu komputasi yang cukup lama untuk menentukan struktur PNN yang baik. Makalah ini menjelaskan usaha perbaikan kinerja waktu proses implementasi SPNN dengan memperhatikan bagian-bagian proses yang *independent* serta memodifikasi algoritmanya untuk dapat diterapkan pemrosesan secara paralel. Hasil eksperimen menunjukkan percepatan yang cukup berarti.

Kata kunci : *probabilistik neural network*, *neural network*, *klasifikasi pola*

1. Pendahuluan

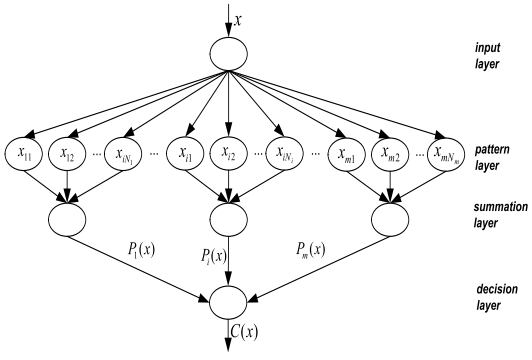
1.1. Latar Belakang

Probabilistic Neural Network (PNN) adalah salah satu tipe jaringan neural yang umum digunakan untuk memecahkan permasalahan klasifikasi pola. PNN memiliki struktur jaringan dan metode pelatihan yang sederhana tetapi PNN memiliki kelemahan utama yaitu dalam menentukan struktur jaringan yang terdiri dari penentuan nilai parameter *smoothing* dan jumlah neuron yang digunakan pada lapisan pola. Nilai parameter *smoothing* akan mempengaruhi tingkat keberhasilan klasifikasi, sedangkan jumlah neuron akan mempengaruhi biaya komputasi yang diperlukan dalam proses klasifikasi. Penentuan struktur jaringan ini tidaklah mudah karena tergantung pada data yang digunakan.

Dengan adanya kelemahan ini, maka diajukan algoritma *Supervised* PNN *Structure Determination* (SPNN) dengan tujuan untuk mempermudah penentuan struktur PNN [1]. Akan tetapi, dalam implementasi iteratif, SPNN ini masih memerlukan waktu komputasi yang cukup lama untuk menentukan struktur PNN yang baik [2]. Makalah ini menjelaskan perbaikan algoritma SPNN sehingga menjadi algoritma paralel SPNN dengan tujuan untuk menekan waktu komputasi.

Berikut akan dijelaskan secara singkat mengenai PNN dan SPNN, serta implementasi iteratif dan paralel dari SPNN.

PNN adalah salah satu jaringan neural yang diperkenalkan oleh Donald Specht pada tahun 1990. Jaringan neural ini dirancang menggunakan ide dari teori klasik probabilitas yaitu klasifikasi Bayesian dan *Estimator* PDF Parzen. Struktur PNN terdiri dari empat lapisan seperti pada Gambar 1.



Gambar 1. Arsitektur PNN [1]

Metode pelatihan PNN cukup sederhana, yaitu hanya dengan memasukkan setiap neuron ke dalam lapisan pola dimana satu neuron mewakili satu vektor data pelatihan. Akibatnya, semakin banyak

data pelatihan, semakin banyak pula jumlah neuron pada lapisan pola. Hal ini mengakibatkan biaya komputasi yang semakin tinggi pula.

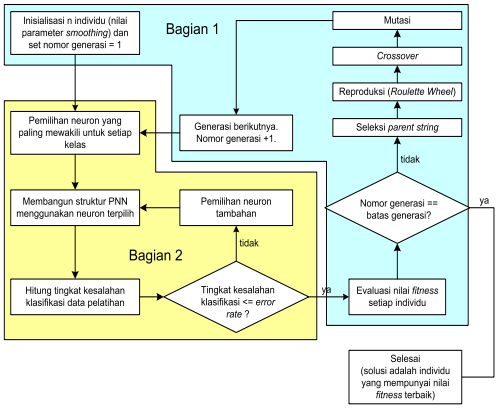
Untuk menghitung nilai aktivasi pada setiap neuron di lapisan pola, diperlukan sebuah parameter yang disebut sebagai parameter *smoothing*. Parameter ini akan mempengaruhi tingkat keberhasilan klasifikasi dengan nilai yang optimal tergantung pada data yang digunakan. Sampai saat ini, belum ada metode matematis yang bisa digunakan untuk mencari nilai optimal dari parameter *smoothing*. Sehingga dalam penentuannya, peneliti sering menggunakan pencarian dengan metode *trial and error*.

Kedua hal di atas yakni jumlah neuron di lapisan pola dan nilai parameter *smoothing*, menjadi permasalahan utama pada struktur PNN. Masalah ini yang kemudian coba diatasi dengan algoritma SPNN.

SPNN dirancang untuk mengatasi masalah penentuan struktur PNN. Algoritma ini terdiri dari dua bagian yang saling berkaitan dan dilaksanakan secara iteratif. Bagian pertama adalah menentukan nilai parameter *smoothing*. Pada bagian ini digunakan algoritma genetika untuk mencari nilai yang optimal. Bagian kedua adalah seleksi neuron-neuron yang mewakili setiap kelas di mana jumlah yang digunakan adalah seminimum mungkin tetapi mampu mengenali seluruh data pelatihan setiap kelas. Alur dari algoritma SPNN dapat dilihat pada Gambar 2.

Pada SPNN, sebuah nilai parameter *smoothing* akan menjadi sebuah individu dimana kromosom dari individu tersebut adalah representasi *string* nilai parameter *smoothing*. Apabila terdapat sebuah nilai parameter *smoothing* 0,4325 (pada penelitian ini digunakan ketelitian 4 angka di belakang koma), maka kromosom dari individu tersebut adalah:

Bit1	Bit2	Bit3	Bit4
4	3	2	5



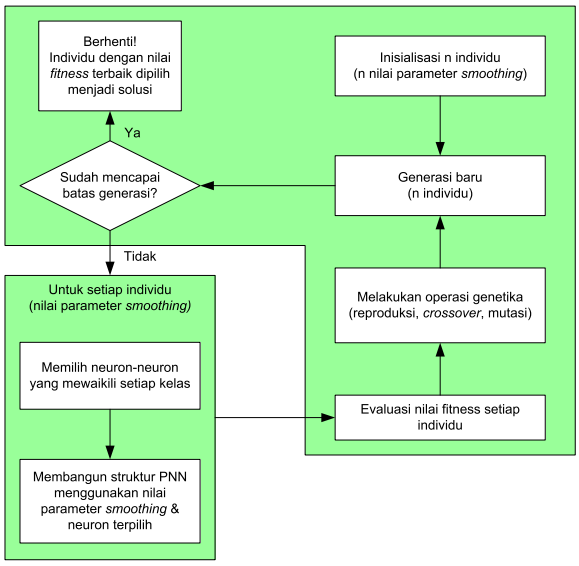
Gambar 2. Alur Algoritma *Supervised* PNN *Structure Determination* (SPNN).

Representasi inilah yang akan digunakan dalam tahap-tahap algoritma genetika, yaitu tahap reproduksi, *crossover*, dan mutasi, untuk mencari nilai parameter *smoothing* yang paling optimal.

Setelah nilai parameter *smoothing* ditentukan, maka selanjutnya adalah menyeleksi neuron-neuron yang terdapat pada lapisan pola untuk meminimalkan ukuran jaringan. Untuk menyeleksi neuron, digunakan algoritma ortogonal. Dengan algoritma ini, setiap kelas akan dipilih sebuah neuron terbaik berdasarkan kriteria tertentu. Jumlah neuron dari setiap kelas pada lapisan pola akan terus ditambah sampai parameter minimum tingkat pengenalan data pelatihan terpenuhi.

1.2. Implementasi Iteratif SPNN

Implementasi iteratif SPNN secara mendasar adalah melakukan seluruh komputasi pada sebuah prosesor dengan alur algoritma dapat dilihat pada Gambar 3.



Gambar 3. Alur implementasi iteratif SPNN.

Implementasi iteratif dari algoritma SPNN secara rinci dapat dilihat pada *pseudocode* di Tabel 1 yang penjelasan dari *pseudocode* tersebut adalah sebagai berikut:

- Fungsi `SearchBestPNNStructure()` adalah fungsi yang dipanggil untuk mencari struktur PNN terbaik dengan parameter :
  - `max_generation` = maksimum jumlah generasi (diperlukan dalam algoritma genetika)
  - `training_data` = data pelatihan PNN
  - `total_individu_per_generation` = jumlah individu dari setiap generasi

  - o `minimum_recognition_rate` = minimum tingkat pengenalan data pelatihan yang harus dipenuhi oleh setiap struktur PNN
- Pertama, bangkitkan sejumlah individu yang menjadi populasi dari generasi pertama dengan menggunakan fungsi `GenerateFirstGeneration()`.
- Kedua, cari struktur minimum dari setiap individu yang telah dihasilkan pada tahap sebelumnya.
- Ketiga, hitung nilai *fitness* dari setiap individu yang sebelumnya harus dicari terlebih dahulu ukuran maksimum dan minimum jaringan dari seluruh individu yang ada.
- Keempat, bangkitkan sejumlah individu baru berdasarkan individu lama dengan menggunakan operasi-operasi dalam algoritma genetika yang dalam tahap ini digunakan nilai *fitness* untuk memilih individu-individu yang baik untuk dijadikan *parent* dalam proses reproduksi.
- Kelima, cari struktur minimum dari setiap individu baru yang telah dihasilkan oleh algoritma genetika.

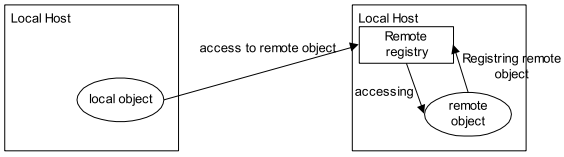
Tabel 1. *Pseudocode* Implementasi Iteratif algoritma SPNN

```
SearchBestPNNStructure(max_generation,training_dat
a,
      total_individu_per_generation,
    minimum_recognition_rate) {

    population=GenerateFirstGeneration(total_indiv
idu_per_generation)

    FindMinimumStructureForEachIndividu(population
,training_data,
        minimum_recognition_rate)

     EvaluateEachIndividu(population)

    for generationNumber=1 until max_generation {
      new_population =
GenerateNewIndividuUsingGeneticAlgorithm(populatio
n)


FindMinimumStructureForEachIndividu(population,
      training_data, minimum_recognition_rate)

        EvaluateEachIndividu(population,
new_population)

        Population = SelectIndividu(population,
new_population,
               total_individu_per_generation)
    }
    solution = SelectBestIndividu(population)

    return solution
}
```

- Keenam, hitung nilai *fitness* dari setiap individu lama dan baru yang sebelumnya harus dicari terlebih dahulu ukuran maksimum dan minimum jaringan dari seluruh (baru dan lama) individu yang ada.
- Ketujuh, pilih sejumlah individu terbaik berdasarkan nilai *fitness* untuk dijadikan populasi generasi selanjutnya.
- Kedelapan, tambah satu nomor generasi. Apabila belum mencapai generasi maksimum, maka kembali ke tahap empat. Jika sudah, maka individu dengan nilai *fitness* terbaik akan dipilih menjadi solusi.

1.3 Java Remote Method Invocation

Perangkat bantu untuk melakukan percobaan implementasi paralel adalah Java *RMI* yang dibuat berdasarkan ide dasar dari *local objects* dan *remote objects*. Konsep ini bersifat relatif. *Local objects* adalah objek-objek yang dijalankan pada *host* tertentu sedangkan *remote objects* adalah objek-objek yang dijalankan pada *host* yang lain. Objek-objek pada *remote host* diekspor (*exported*) sehingga dapat diminta (*invoked*) secara *remote*. Suatu objek diekspor dengan cara melakukan registrasi dengan suatu *remote registry server*. Server *remote registry* membantu objek pada *host* yang lain untuk mengakses secara remote objek yang teregister oleh server itu. Hal itu dilakukan dengan memelihara *database* nama dan objek yang terasosiasi dengan nama itu. [3,4,5]



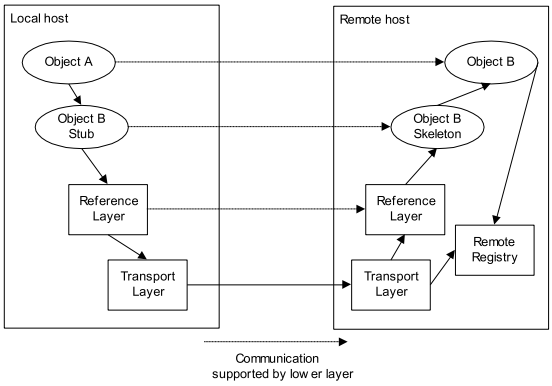
Gambar 4. Registrasi dari suatu *remote object* untuk *remote access* [3]

Objek yang mengekspor dirinya sendiri untuk akses secara *remote* harus mengimplementasi *remote interface*. *Interface* ini mengidentifikasi objek agar dapat diakses secara *remote*. Setiap metode yang diminta secara *remote* harus "membuang" (*throw*) *Remote Exception*. Eksepsi ini digunakan untuk mengindikasi kesalahan yang terjadi selama proses RMI.

Metode pendekatan RMI pada Java (Java RMI) adalah objek-objek diorganisir dalam suatu *client/server framework*. Suatu *local object* yang meminta (*invoke*) metode dari *remote object* dijadikan sebagai *client object* atau *client*. Sebuah *remote object* yang metodenya diminta oleh *local object* dijadikan sebagai sebagai *server object* atau *server*.

Java RMI menggunakan *stub* dan *skeleton* untuk mengakses *object remote*. *Stub* adalah suatu *local object* yang berlaku sebagai *local proxy* untuk *remote object*. *Stub* menyediakan metode yang sama seperti *remote object*. *Local object* meminta (*invoke*) metode dari *stub* seakan seperti metode dari *remote object*. *Stub* kemudian mengkomunikasikan permintaan (*invocation*) metode kepada *remote object* melalui suatu *skeleton* yang diimplementasikan di *remote host*. *Skeleton* adalah suatu *proxy* pada *remote object* yang berada pada *host* yang sama dengan *remote object*. *Skeleton* berkomunikasi dengan local *stub* dan menyampaikan invokasi metode pada *stub* kepada *remote object* yang sebenarnya. *Skeleton* kemudian menerima nilai yang dihasilkan dari RMI (jika ada) dan menyampaikan nilai itu kembali ke *stub*. *Stub* selanjutnya mengirim nilai tersebut kepada *local object* yang menginisiasi RMI.

Stub dan *skeleton* berkomunikasi melalui *remote reference layer*. *Layer* ini menyediakan *stub* dengan kemampuan untuk berkomunikasi dengan *skeleton* melalui suatu protokol *transport*. RMI biasanya menggunakan TCP untuk *transport* informasi, walaupun bersifat fleksibel untuk dapat menggunakan protokol lainnya.



Gambar 5. Java RMI menggunakan *stub* dan *skeleton* untuk mensupport komunikasi *client* / *server* [3].

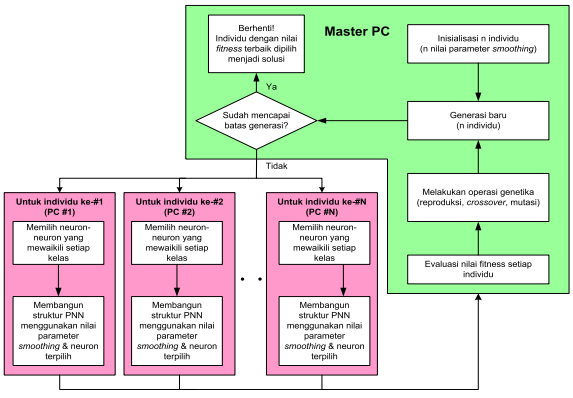
Untuk membuat aplikasi terdistribusi yang berkomunikasi dengan teknik, maka pada setiap komputer *server* dan *client* cukup di-*install* Java Development Kit (JDK). Rincian bagaimana ini dapat diimplementasikan dapat dilihat pada bahan di daftar pustaka.

2. Metode Penelitian

Dalam penelitian di makalah ini, algoritma iteratif SPNN dikaji dengan melihat bagian yang paling memakan waktu. Kemudian dilihat kemungkinan pemrosesan pada bagian tersebut dibagi-bagi menjadi subbagian yang dapat diproses secara terpisah. Dari sini, algoritma paralel dan dilakukan implementasi dengan Java RMI dikembangkan.

Implementasi paralel dari SPNN bertujuan untuk menekan waktu komputasi yang diperlukan pada implementasi iteratif. Pada penelitian ini, digunakan teknik *Message Passing Interface* untuk mendistribusikan tugas komputasi ke sejumlah komputer sehingga diharapkan waktu akan lebih rendah dibandingkan hanya menggunakan sebuah komputer. Gambar 3 telah memperlihatkan alur SPNN pada implementasi iteratif. Alur SPNN dalam implementasi paralel dapat dilihat pada Gambar 6.

Pada bagian sebelumnya telah dijelaskan bahwa SPNN terbagi menjadi dua bagian yaitu mencari nilai optimal parameter *smoothing* dan seleksi neuron-neuron pada lapisan pola. Dalam penelitian ini, pendistribusian komputasi dilakukan dengan mendistribusikan komputasi pada bagian kedua (seleksi neuron) ke sejumlah komputer. Karena berdasarkan percobaan bagian inilah yang memerlukan waktu komputasi jauh lebih lama (lebih dari 95%) dibandingkan komputasi pada bagian kedua.



Gambar 6. Alur implementasi paralel SPNN.

Struktur sistem yang digunakan dalam implementasi paralel juga dapat dilihat pada Gambar 6. Sebuah komputer akan menjadi *master* PC yang mempunyai tugas untuk melakukan komputasi bagian pertama (mencari nilai optimal parameter *smoothing*) dan mendistribusikan tugas komputasi bagian kedua ke satu atau lebih *slave* PC dengan memberikan data pelatihan dan nilai parameter *smoothing* untuk setiap komputer.

Karena data pelatihan yang digunakan pada setiap komputasi bagian kedua adalah sama, maka pendistribusian data ke *slave* PC hanya dilakukan satu kali yaitu pada awal proses pencarian. Untuk komunikasi antara *master* dan *slave* PC, digunakan metode *Remote Method Invocation* yang telah tersedia pada pemrograman bahasa Java. Tabel 2 menampilkan *pseudocode* dari program yang dijalankan pada *master* PC, sedangkan Tabel 3 menampilkan *pseudocode* dari program yang dijalankan pada *slave* PC.

Apabila *pseudocode* di Tabel 3 tersebut dibandingkan dengan *pseudocode* pada implementasi iteratif, dapat dilihat bahwa perbedaan utama terletak pada fungsi pencarian minimum struktur jaringan di mana komputasi dilakukan pada sebuah komputer, sedangkan pada implementasi paralel, komputasi pencarian minimum didistribusikan ke sejumlah komputer yang kemudian hasilnya dikumpulkan untuk digunakan pada komputasi algoritma genetika.

3. Percobaan

Bagian ini menjelaskan desain, hasil, dan analisis percobaan yang telah dilakukan. Percobaan dilakukan dengan menggunakan empat buah *Personal Computer* (PC) dengan spesifikasi ID, Processor dan Memori masing-masing adalah A, Intel Centrino 1.3 Ghz, 256 MB ; B, Intel Pentium III 1.7 Ghz, 256 MB ; C, Intel Pentium III 1 Ghz, 128 MB ; dan D, Intel Pentium III 1 Ghz, 128 MB.

Tabel 2. *Pseudocode* program yang dijalankan oleh *Master*

```
SearchBestPNNStructure(total_generation,training
_data,
     total_individu_per_generation,
minimum_recognition_rate) {
     SetSlaveParameter(training_data,
minimum_recognition_rate)

     Population=GenerateFirstGeneration(total_in
dividu_per_generation)

     /* assign the task to find minimum
structure of each individu
        to slave computer and collect the result
*/

     DistributeIndividuToSlaveComputer(populatio
n,

     training_data,minimum_recognition_rate)
     EvaluateEachIndividu(population)

for generationNumber=1 until total_generation {
      new_population =
GenerateNewIndividuUsingGeneticAlgorithm(populat
ion)

      /* assign the task to find minimum
structure of each individu
           to slave computer and collect the
result */

     DistributeIndividuToSlaveComputer(populatio
n,minimum_recognition_rate)
     EvaluateEachIndividu(population,
new_population)
     population =
SelectIndividu(population,new_population,
                total_individu_per_generation)
     }
     solution = SelectBestIndividu(population)

     return solution
```

Tabel 3. *Pseudocode* yang dijalankan oleh *Slave/Server*

```
  setNextTask(smoothing_parameter,
       minimum_recognition_rate ) {
  FindMinimumStructureForEachIndividu(population,
  minimum_recognition_rate)
    SendSizeOfStructureToMasterPC()
  }

  SetParameter(training_data,
  minimum_recognition_rate) {
    local_training_data = training_data
    local_minimum_recognition_rate=
  minimum_recognition_rate
  }
```

Setelah dilakukan beberapa percobaan menggunakan implementasi iteratif, ternyata PC-A dapat menyelesaikan komputasi dengan waktu yang lebih rendah dibandingkan dengan PC-B. Karena itu, waktu komputasi dari implementasi iteratif pada PC-A akan dijadikan acuan untuk mengukur penurunan waktu komputasi pada implementasi paralel.

Pada percobaan ini, digunakan data percobaan pengenalan aroma dua campuran antara jeruk dengan alkohol yang terdiri dari kelas-kelas berikut ini :

1. Campuran jeruk dengan alkohol 0%
2. Campuran jeruk dengan alkohol 15%
3. Campuran jeruk dengan alkohol 25%
4. Campuran jeruk dengan alkohol 35%
5. Campuran jeruk dengan alkohol 45%
6. Campuran jeruk dengan alkohol 75%

Jumlah total kelas data pelatihan adalah 6 kelas yang pada setiap kelas akan digunakan 100 vektor data pelatihan sehingga total seluruh data pelatihan adalah 600 vektor data.

Pada percobaan akan dilakukan 4 kali untuk setiap set parameter, yaitu:

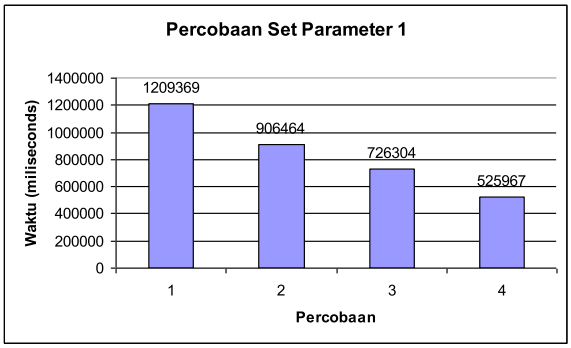
1. Implementasi iteratif menggunakan menggunakan 1 PC
2. Implementasi paralel menggunakan 1 *master* PC + 2 *slave* PC
3. Implementasi paralel menggunakan 1 *master* PC + 3 *slave* PC
4. Implementasi paralel menggunakan 1 *master* PC + 4 *slave* PC

Untuk percobaan keempat (1 *master* PC + 4 *slave* PC), komputer yang dijadikan sebagai *master* PC juga akan dijadikan *slave* PC dengan menjalankan program *master* dan *slave* di komputer tersebut. Selain itu, dalam percobaan ini juga akan digunakan 2 set parameter, yaitu :

1. jumlah generasi = 20, jumlah individu per generasi = 10, minimum tingkat pengenalan data pelatihan = 1.0
2. jumlah generasi = 20, jumlah individu per generasi = 20, minimum tingkat pengenalan data pelatihan = 1.0

4. Hasil dan Pembahasan

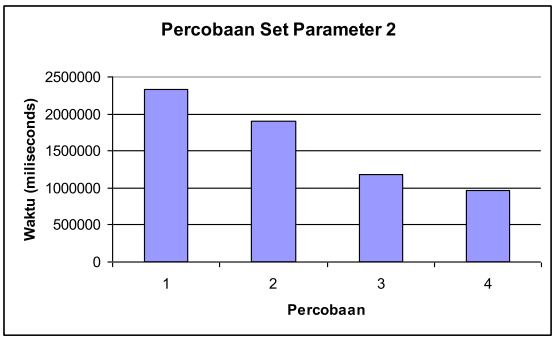
Hasil percobaan menggunakan set parameter 1 dapat dilihat pada Gambar 7. Pada Gambar 7, notasi (1) menyatakan percobaan iteratif, (2) percobaan 1 *master* dan 2 *slave*, (3) percobaan dengan 1 *master* dan 3 *slave* serta (4) percobaan dengan 1 *master* dan 4 *slave*.



Gambar 7. Grafik waktu komputer (*miliseconds*) dengan set parameter 1.

Gambar 7 menunjukkan bahwa terjadi penurunan waktu komputasi implementasi paralel bila dibandingkan dengan implementasi iteratif. Namun angka penurunan dari setiap konfigurasi tidak mengikuti suatu pola tertentu karena perbedaan spesifikasi komputer yang digunakan.

Hasil percobaan menggunakan set parameter 2 dapat dilihat pada Gambar 8. Notasi (1), (2), (3), dan (4) menyatakan percobaan dengan jumlah *master* dan *slave* yang sama seperti pada Gambar 7.



Gambar 8. Grafik waktu komputer (*miliseconds*) percobaan dengan set parameter 2.

Hasil yang hampir sama juga dicapai pada percobaan menggunakan set parameter 2, yaitu terjadi penurunan waktu komputer implementasi paralel bila dibandingkan dengan implementasi iteratif. Dari kedua grafik tersebut dapat dilihat juga bahwa penambahan jumlah komputer dapat semakin menurunkan waktu komputasi.

5. Kesimpulan

Dalam penelitian ini dapat dilihat bahwa implementasi paralel dari SPNN dapat menekan waktu komputasi yang diperlukan dibandingkan dengan implementasi iteratif. Selain itu, semakin bertambahnya jumlah komputer yang digunakan juga menyebabkan semakin menurunnya waktu komputasi yang diperlukan. Akan tetapi, masih perlu dilakukan percobaan lain untuk melihat batas maksimal penambahan komputer terhadap efek penurunan waktu komputasi menggunakan implementasi paralel ini. Penurunan waktu antara lain dipengaruhi oleh komunikasi antar-*processor* yang terlibat.

REFERENSI

[1] Mao, K.Z., dkk. "Probabilistic neural-network structure determination for pattern classification", *IEEE Transaction On Neural Networks*, Volume 11 No. 4 July 2000.

[2] Herry. "Kinerja PNN-teroptimasi berbasis algoritma genetika dalam pengenalan aroma 2 campuran", Laporan Tugas Akhir, Fakultas Ilmu Komputer Universitas Indonesia (Fasilkom UI), 2002.

[3] Jaworski, Jamie, *Java™ 2 Platform Unleashed* (Sams, 1999).

[4] Chaffee, Alexander Day, Java Remote Method Invocation (RMI) *http://www.purpletech.com/talks.jsp/RMI.ppt* (terakhir diakses 15 Mei 2006) .
[5] Suhartanto, Heru, Bahan Mata Kuliah Komputasi Tersebar, Fasilkom UI, 2004.